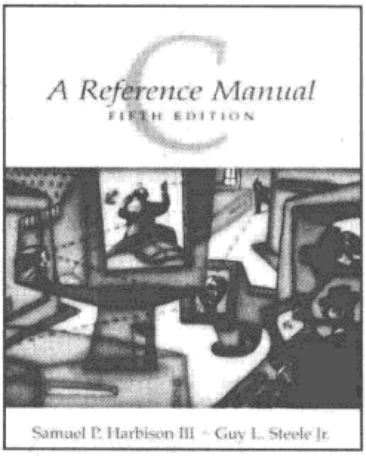
C语言经典译丛

C语言参考手册

原书第5版



C: A Reference Manual

(Fifth Edition)

美） Samuel P. Harbison III
Guy L. Steele Jr. 著

徐波 等译

机械工业出版社
China Machine Press

译 者 序

在众多讲述C语言的书籍中，本书是一部非常优秀的作品。它详细、精确地描述了C语言所有方面的特性。说它是C程序员案头必备的参考手册，绝对不是夸张。

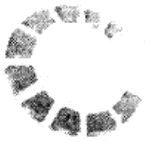
本书的作者都是享有盛誉的C语言专家，并且都曾担任C/C++语言标准委员会的成员，对C语言有着深入、独到的理解。本书正是他们殚精虑竭之作。这本手册技术细节描述精确，组织条理清晰，内容完备详尽而又简明扼要，在广度、深度和精度方面都是出类拔萃的。并且经过5个版本的完善之后，本书已经是一部非常成熟的作品。

本书并不是一本指导初学者从头学习C语言的教程，而是一本供专业程序员在开发程序时随时查阅的参考书。它对C语言各个方面的描述既精确又扼要，对于C语言有关的任何问题，几乎都可以在本书中找到答案。它所涵盖的C语言版本包括传统C、C89、C89修正案1和C99，尤其是对C99的详细介绍，是其他C语言名作所不具备的。因此，不管读者所使用的是什么类型的C编译器，都能够从本书中受益。另外，本书还专门注意了C和C++所存在的兼容性问题，对于想开发C/C++兼容的应用程序的开发人员而言，尤其具有价值。难怪有专家评论，除了Brian W. Kernighan和Dennis M. Ritchie的经典名作《The C Programming Language》[⊖]之外，唯一离不开的就是这本书。

本书分为两个部分，第一部分是C语言本身的描述，第二部分则介绍了标准函数库。尽管本书介绍了C语言的完整知识，但它的篇幅却不是非常庞大，这一方面是由于C语言本身比较简洁，另一方面也是由于本书所使用的语言非常精练。对于专业程序员而言，最好的参考手册就是能够准确而精练地描述问题，并提供适当的例子。本书就是一本完全符合这个条件的参考手册。

参加本书翻译工作的人员有徐波、陈永军、李福军、毛玉山、冯毅晓、陈丽娜、杨洁、张瑜、应巧敏、顾沂、董志杰、许朝阳、龚建峰等。

⊖ 该书的中译本和影印版均已由机械工业出版社引进出版，书名为《C程序设计语言（英文版·第2版）》(978-7-111-19626-0)，《C程序设计语言（第2版·新版）》(978-7-111-12806-0)。——编辑注



前　言

本书是C编程语言的参考手册。本书的目标是对C语言的基本概念和运行时函数库提供完整而准确的描述，同时还强调了以正确性、可移植性和可维护性为基本出发点的良好编程风格。

本书预期读者已经理解基本的编程概念，并希望很多读者已经可以用C语言熟练编程。为了保持参考手册的格式，本书从头到尾介绍C语言的词法结构、预处理器、声明、类型、表达式、语句、函数和运行时函数库。书中包括了许多交叉引用，使读者可以从任何地方入手。

本版完整地介绍了最新的C语言国际标准ISO/IEC 9899:1999（C99），明确地指出了语言本身和库函数的哪些特性是C99新增的，并指出了C99与原有的C89标准的不同之处。这是目前唯一适用于所有主流C语言版本的参考手册，涵盖了传统C、1989年C标准、1995年对C89的修改和补充以及最新的C99标准。本书还介绍了标准C和标准C++的公共子集Clean C。尽管C99增加了许多新信息，但本书并没有对章节结构进行很大的改动，这样熟悉旧版的读者可以顺利地找到他所需要的材料。

本书最初源于我们在Tartan公司的工作，为从微机到大型机的一系列计算机开发C语言编译器系列。我们要求编译器文档齐全，提供精确而有用的错误诊断信息，并能产生性能卓越的目标代码。经过某个编译器正确编译的C程序应该能够在硬件差别允许的情况下，在其他所有编译器中也能顺利通过编译。

1984年，尽管C语言已经非常普及，但还没有一本书对C语言的介绍精确到足以指导我们设计新的编译器。另外，当时的语言描述对于那些想使用编译器对C程序进行深入分析的编程人员和客户而言仍然不够精确。在本书中，我们特别注重那些影响程序的清晰度、目标代码的有效性以及不同环境中程序移植性的语言特性。

Web站点

读者可以访问本书的Web站点CAReferenceManual.com，其中包括本书的示例代码、更深入的讨论、澄清的问题以及C语言资源的更多链接。

致谢

在准备本书的第5版时，特别感谢原NCITS JII主席Rex Jaeschke、芬兰赫尔辛基的Antoine Trux以及爱迪生设计集团创始人Steve Adamczyk的帮助。

对于本书以前的版本提供过帮助的人员包括Jeffrey Esakov、Alan J.Filipski、Frank J.Wagner、Debra Martin、P.J.Plauger和Steve Vinoski。其他提供过帮助的人员包括Aurelio Bignoli、Steve

Clamage、Arthur Evans、Jr.、Roy J. Fuller、Morris M.Kessan、George V.Reilly、Mark Lan、Mike Hewett、Charles Fischer、Kevin Rodgers、Tom Gibb、David Lim、Stavros Macrakis、Steve Vegdahl、Christopher Vickery、Peter van der Linden和Dave Wilson。还要感谢Machael Angus、Mady Bauer、Larry Breed、Sue Broughton、Alex Czajkowski、Robert Firth、David Gaffney、Steve Gorman、Dennis Hamilton、Chris Hanna、Ken Harrenstien、Rex Jaeschke、Don Lindsay、Tom MacDonald、Peter Nelson、Joe Newcomer、Kevin Nolish、David Notkin、Peter Plamondon、Roger Ray、Larry Rosler、David Spencer和Barbara Steele。

本书最初的一些示例程序参考了下列著作中的算法：

- Beeler，Michael，Gosper，R.William和Schroeppel，Richard，《HAKMEM》，AI Memo 239（麻省理工学院人工智能实验室，1972年2月）。
- Bentley，Jon Louis，《Writting Efficient Programs》（Prentice-Hall，1982）。
- Bentley，Jon Louis，《Programming Pearls》（1983年8月起出现在《Communications of the ACM》的每月专栏）。
- Kernighan，Brian W.和Ritchie，Dennis M.《The C Programming Language》（Prentice-Hall，1978）。
- Knuth，Donald E.《The Art of Computer Programming》[⊖]第1-3卷（Addison-Wesley，1968、1969、1973、1981）。
- Sedgewick，Robert，《Algorithms》[⊜]（Addison-Wesley，1983）。

感谢这些作者的灵感。

还有一点需要说明，Guy Steele先生由于工作繁忙，无法在新版本中参与更多工作。虽然书中仍然处处体现出他对C语言的精僻分析，但新版本所存在的任何问题，都不应该再由他负责。

《C语言参考手册》一书已经出版了很多年，感谢所有读者多年来对它的关爱。

Sam Harbison
于宾夕法尼亚州匹兹堡市
harbison@CAReferenceManual.com

⊖ 该书影印版已由机械工业出版社引进出版，书名为《计算机程序设计艺术 第1卷 基本算法（英文版·第3版）》（978-7-111-22709-0）、《计算机程序设计艺术 第2卷 半数值算法（英文版·第3版）》（978-7-111-22718-2）和《计算机程序设计艺术 第3卷 排序和查找（英文版·第2版）》（978-7-111-22717-5）。——编辑注

⊜ 该书影印版已由机械工业出版社引进出版，书名为《算法：C语言实现（第1～4部分）基础知识、数据结构、排序及搜索（英文版·第3版）》（978-7-111-19764-X）和《算法：C语言实现（第5部分）图算法（英文版·第3版）》（978-7-111-19769-0）。——编辑注

本书是C语言的详尽参考手册，分为两部分：第一部分讨论了C语言的所有语言特征，包括词法、预处理机制、声明、类型、表达式、语句以及函数等基本语言特征；第二部分讨论了C语言的标准库，针对它们不同的功能分别进行详细介绍。为了帮助读者理解相关概念，本书在讨论C语言及其标准库的细节时，提供了许多实例和解释。第一部分的各章还提供了练习题和主要练习的解答，这些练习可帮助读者加深理解C语言的基本功能和特性。

本书可以作为高等院校计算机专业C语言课程的教材和参考书，也可供程序开发人员参考。

Authorized translation from the English language edition, entitled C: A Reference Manual, 5th Edition, 0-13-089592-X by Samuel P. Harbison III, Guy L. Steele Jr., published by Pearson Education, Inc., publishing as Addison Wesley Professional, Copyright © 2002 .

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc..

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD., and CHINA MACHINE PRESS Copyright © 2011.

本书封面贴有Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

封底无防伪标均为盗版
版权所有，侵权必究
本书法律顾问　北京市展达律师事务所

本书版权登记号：图字：01-2002-3650

图书在版编目（CIP）数据

C语言参考手册（原书第5版）/（美）哈比森（Harbison, S. P.）等著；徐波等译. —北京：机械工业出版社，2011.6
（C语言经典译丛）
书名原文：C: A Reference Manual, Fifth Edition

ISBN 978-7-111-34684-5

Ⅰ. C…　Ⅱ. ① 哈…　② 徐…　Ⅲ. C语言—程序设计—高等学校—教学参考资料　Ⅳ. TP312

中国版本图书馆CIP数据核字（2011）第088147号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）
责任编辑：秦　健
北京市荣盛彩色印刷有限公司印刷
2011年6月第1版第1次印刷
186mm × 240mm · 26印张
标准书号：ISBN 978-7-111-34684-5
定价：69.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991；88361066
购书热线：（010）68326294；88379649；68995259
投稿热线：（010）88379604
读者信箱：hzjsj@hzbook.com



目　录

译者序
前　言

第一部分　C语言

第1章　概述 ……2
1.1　C语言的演变 ……2
1.1.1　传统C ……2
1.1.2　标准C（1989） ……2
1.1.3　标准C（1995） ……3
1.1.4　标准C（1999） ……3
1.1.5　标准C++ ……4
1.1.6　本书内容 ……4
1.2　应该使用哪种C语言 ……4
1.3　C编程概述 ……5
1.4　一致性 ……6
1.5　语法约定 ……7
第2章　词法元素 ……8
2.1　字符集 ……8
2.1.1　执行字符集 ……9
2.1.2　空白字符和行终止符 ……9
2.1.3　字符编码 ……10
2.1.4　三字符组 ……10
2.1.5　多字节字符和宽字符 ……11
2.2　注释 ……13
2.3　标记 ……14
2.4　操作符和分隔符 ……15
2.5　标识符 ……15
2.6　关键字 ……17
2.7　常量 ……18
2.7.1　整数常量 ……18
2.7.2　浮点数常量 ……22
2.7.3　字符常量 ……23
2.7.4　字符串常量 ……25
2.7.5　转义字符 ……27
2.7.6　字符转义码 ……28
2.7.7　数值转义码 ……29
2.8　C++兼容性 ……29
2.8.1　字符集 ……30
2.8.2　注释 ……30
2.8.3　操作符 ……30
2.8.4　标识符和关键字 ……30
2.8.5　字符常量 ……30
2.9　关于字符集、指令集和编码 ……30
2.10　练习 ……32
第3章　C预处理器 ……34
3.1　预处理器命令 ……34
3.2　预处理器词法约定 ……35
3.3　定义和替换 ……36
3.3.1　类似对象的宏定义 ……36
3.3.2　定义带参数的宏 ……37
3.3.3　宏表达式的重新扫描 ……39
3.3.4　预定义的宏 ……40
3.3.5　取消宏定义和重新定义宏 ……42
3.3.6　宏展开的优先级错误 ……42
3.3.7　宏参数的副作用 ……43
3.3.8　把标记转换为字符串 ……43
3.3.9　宏展开中的标记合并 ……44
3.3.10　宏的可变参数列表 ……45
3.3.11　其他问题 ……46
3.4　文件包含 ……46
3.5　条件编译 ……48
3.5.1　#if、#else和#endif命令 ……48
3.5.2　#elif命令 ……49
3.5.3　#ifdef和#ifndef命令 ……50

3.5.4 条件命令中的常量表达式 ……51
3.5.5 defined操作符 ……52
3.6 显式的行号 ……52
3.7 pragma指令 ……53
3.7.1 标准pragma命令 ……53
3.7.2 标准pragma指令的位置 ……54
3.7.3 _Pragma操作符 ……54
3.8 错误指令 ……54
3.9 C++兼容性 ……55
3.10 练习 ……55
第4章 声明 ……57
4.1 声明的组织形式 ……57
4.2 术语 ……58
4.2.1 作用域 ……58
4.2.2 可见性 ……59
4.2.3 前向引用 ……59
4.2.4 名称的重载 ……60
4.2.5 重复声明 ……61
4.2.6 重复可见性 ……62
4.2.7 范围 ……62
4.2.8 初始值 ……63
4.2.9 外部名称 ……64
4.2.10 编译时名称 ……65
4.3 存储类别和函数指定符 ……65
4.3.1 默认存储类别指定符 ……66
4.3.2 存储类别指定符的例子 ……66
4.3.3 函数指定符 ……67
4.4 类型指定符和限定符 ……67
4.4.1 默认类型指定符 ……68
4.4.2 缺失的声明器 ……69
4.4.3 类型限定符 ……70
4.4.4 const ……70
4.4.5 volatile和序列点 ……72
4.4.6 restrict ……74
4.5 声明器 ……75
4.5.1 简单声明器 ……75
4.5.2 指针声明器 ……76
4.5.3 数组声明器 ……77
4.5.4 函数声明器 ……78
4.5.5 声明器的组合 ……80
4.6 初始化值 ……81
4.6.1 整数 ……82
4.6.2 浮点数 ……83
4.6.3 指针 ……83
4.6.4 数组 ……84
4.6.5 枚举 ……86
4.6.6 结构 ……86
4.6.7 联合 ……87
4.6.8 省略花括号 ……88
4.6.9 指定成员的初始化值 ……88
4.7 隐式声明 ……89
4.8 外部名称 ……90
4.8.1 初始化值模型 ……90
4.8.2 省略的存储类别模型 ……90
4.8.3 公共模型 ……90
4.8.4 混合公共模型 ……91
4.8.5 总结和建议 ……91
4.8.6 未引用的外部声明 ……92
4.9 C++兼容性 ……92
4.9.1 作用域 ……92
4.9.2 标签和typedef名称 ……92
4.9.3 用于类型的存储类别指定符 ……93
4.9.4 const类型限定符 ……93
4.9.5 初始化值 ……93
4.9.6 隐式声明 ……93
4.9.7 定义和引用声明 ……94
4.9.8 函数链接 ……94
4.9.9 无参函数 ……94
4.10 练习 ……94
第5章 类型 ……97
5.1 整数类型 ……98
5.1.1 有符号整数类型 ……98
5.1.2 无符号整数类型 ……101
5.1.3 字符类型 ……102
5.1.4 扩展整数类型 ……104
5.1.5 布尔类型 ……104

5.2 浮点类型 ……104
5.3 指针类型 ……108
5.3.1 通用指针 ……109
5.3.2 null指针和非法指针 ……109
5.3.3 使用指针的一些警告 ……110
5.4 数组类型 ……111
5.4.1 数组和指针 ……111
5.4.2 多维数组 ……112
5.4.3 数组边界 ……112
5.4.4 操作 ……113
5.4.5 可变长度的数组 ……113
5.5 枚举类型 ……115
5.6 结构类型 ……117
5.6.1 结构类型引用 ……119
5.6.2 结构上的操作 ……120
5.6.3 成员 ……120
5.6.4 结构成员的布局 ……122
5.6.5 位段 ……122
5.6.6 移植性问题 ……124
5.6.7 结构的大小 ……125
5.6.8 灵活数组成员 ……126
5.7 联合类型 ……127
5.7.1 联合成员的布局 ……127
5.7.2 联合的大小 ……128
5.7.3 使用联合类型 ……129
5.7.4 （其他）使用联合类型 ……130
5.8 函数类型 ……131
5.9 void类型 ……133
5.10 typedef名称 ……133
5.10.1 用于函数类型的typedef名称 ……135
5.10.2 重定义typedef名称 ……135
5.10.3 编译器的说明 ……136
5.11 类型兼容性 ……136
5.11.1 等价的类型 ……136
5.11.2 枚举兼容性 ……137
5.11.3 数组兼容性 ……138
5.11.4 函数兼容性 ……138
5.11.5 结构和联合兼容性 ……139
5.11.6 指针兼容性 ……139
5.11.7 源文件之间的兼容性 ……139
5.12 类型名称和抽象声明器 ……139
5.13 C++兼容性 ……141
5.13.1 枚举类型 ……141
5.13.2 typedef名称 ……141
5.13.3 类型兼容性 ……141
5.14 练习 ……141
第6章 转换和表示形式 ……144
6.1 表示形式 ……144
6.1.1 存储单位和数据长度 ……144
6.1.2 字节顺序 ……145
6.1.3 对齐限制 ……146
6.1.4 指针的长度 ……147
6.1.5 地址模型的效果 ……147
6.1.6 类型表示形式 ……149
6.2 转换 ……150
6.2.1 表示形式的变化 ……150
6.2.2 细微的转换 ……150
6.2.3 转换为整数类型 ……150
6.2.4 转换为浮点类型 ……152
6.2.5 转换为结构和联合类型 ……152
6.2.6 转换为枚举类型 ……152
6.2.7 转换为指针类型 ……152
6.2.8 转换为数组和函数类型 ……153
6.2.9 转换为void类型 ……153
6.3 寻常转换 ……154
6.3.1 类型转换 ……154
6.3.2 赋值转换 ……154
6.3.3 寻常单目转换 ……155
6.3.4 寻常双目转换 ……157
6.3.5 默认的函数实参转换 ……158
6.3.6 其他函数转换 ……159
6.4 C++兼容性 ……159
6.5 练习 ……159
第7章 表达式 ……161
7.1 对象、左值和指示符 ……161
7.2 表达式和优先级 ……162

7.2.1 操作符的优先级和结合性 ……162
7.2.2 溢出和其他算术异常 ……163
7.3 基本表达式 ……164
7.3.1 名称 ……164
7.3.2 字面值 ……165
7.3.3 带括号的表达式 ……165
7.4 后缀表达式 ……166
7.4.1 下标表达式 ……166
7.4.2 成员选择 ……168
7.4.3 函数调用 ……169
7.4.4 后缀增值和减值操作符 ……171
7.4.5 复合字面值 ……172
7.5 单目表达式 ……173
7.5.1 类型转换 ……174
7.5.2 sizeof操作符 ……174
7.5.3 单目负号和正号 ……176
7.5.4 逻辑反 ……176
7.5.5 位反 ……177
7.5.6 取地址操作符 ……177
7.5.7 间接访问 ……178
7.5.8 前缀增值和减值操作符 ……179
7.6 双目操作符表达式 ……180
7.6.1 乘除操作符 ……180
7.6.2 加减操作符 ……182
7.6.3 移位操作符 ……183
7.6.4 关系操作符 ……185
7.6.5 相等操作符 ……186
7.6.6 位操作符 ……188
7.6.7 整数集合的例子 ……189
7.7 逻辑操作符表达式 ……192
7.8 条件表达式 ……194
7.9 赋值表达式 ……195
7.9.1 简单赋值 ……196
7.9.2 复合赋值 ……197
7.10 线性表达式 ……197
7.11 常量表达式 ……198
7.11.1 预处理器常量表达式 ……199
7.11.2 整型常量表达式 ……200
7.11.3 初始化值常量表达式 ……200
7.12 求值的顺序 ……201
7.13 被丢弃的值 ……202
7.14 内存访问的优化 ……203
7.15 C++兼容性 ……204
7.16 练习 ……204
第8章 语句 ……206
8.1 语句的基本语法规则 ……206
8.2 表达式语句 ……207
8.3 标签语句 ……207
8.4 复合语句 ……208
8.5 条件语句 ……210
8.5.1 多路条件语句 ……210
8.5.2 悬疑的else问题 ……211
8.6 迭代式语句 ……212
8.6.1 while语句 ……212
8.6.2 do语句 ……213
8.6.3 for语句 ……214
8.6.4 使用for语句 ……215
8.6.5 多个控制变量 ……217
8.7 switch语句 ……218
8.8 break和continue语句 ……220
8.9 return语句 ……222
8.10 goto语句 ……223
8.11 空语句 ……224
8.12 C++兼容性 ……224
8.12.1 复合语句 ……224
8.12.2 循环中的声明 ……224
8.13 练习 ……225
第9章 函数 ……226
9.1 函数定义 ……226
9.2 函数原型 ……228
9.2.1 原型出现在什么时候 ……230
9.2.2 混合使用原型和非原型声明 ……230
9.2.3 合理地使用原型 ……231
9.2.4 原型和调用约定 ……231
9.2.5 标准C和传统C的兼容性 ……233
9.3 形式参数声明 ……233

9.4 形参类型的调整 ……235
9.5 参数传递约定 ……236
9.6 形参的一致 ……237
9.7 函数的返回类型 ……238
9.8 返回类型的一致 ……239
9.9 main函数 ……239
9.10 内联函数 ……240
9.11 C++兼容性 ……242
9.11.1 原型 ……242
9.11.2 形参和返回类型中的类型声明 ……242
9.11.3 返回类型的一致 ……242
9.11.4 main ……242
9.11.5 内联 ……243
9.12 练习 ……243

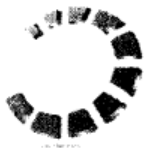
第二部分 C函数库

第10章 函数库简介 ……246
10.1 标准C工具 ……246
10.2 C++兼容性 ……247
10.3 库头文件和名称 ……249
10.3.1 assert.h ……249
10.3.2 complex.h ……249
10.3.3 ctype.h ……249
10.3.4 errno.h ……249
10.3.5 fenv.h ……249
10.3.6 float.h ……250
10.3.7 inttypes.h ……250
10.3.8 iso64.h ……250
10.3.9 limits.h ……250
10.3.10 locale.h ……250
10.3.11 math.h ……251
10.3.12 setjmp.h ……252
10.3.13 signal.h ……252
10.3.14 stdarg.h ……252
10.3.15 stdbool.h ……252
10.3.16 stddef.h ……252
10.3.17 stdint.h ……252
10.3.18 stdio.h ……252
10.3.19 stdlib.h ……253
10.3.20 string.h ……253
10.3.21 tgmath.h ……253
10.3.22 time.h ……254
10.3.23 wchar.h ……254
10.3.24 wctype.h ……254
第11章 标准语言附加 ……255
11.1 NULL、ptrdiff_t、size_t、offsetof ……255
11.2 EDOM、ERANGE、EILSEQ、errno、strerror、perror ……256
11.3 bool、false、true ……257
11.4 va_list、va_start、va_arg、va_end ……258
11.5 标准C的操作符宏 ……261
第12章 字符处理 ……262
12.1 isalnum、isalpha、iscntrl、iswalnum、iswalpha、iswcntrl ……262
12.2 iscsym、iscsymf ……264
12.3 isdigit、isodigit、isxdigit、iswdigit、iswxdigit ……264
12.4 isgraph、isprint、ispunct、iswgraph、iswprint、iswpunct ……265
12.5 islower、isupper、iswlower、iswupper ……266
12.6 isblank、isspace、iswhite、iswspace ……266
12.7 toascii ……267
12.8 toint ……267
12.9 tolower、toupper、towlower、towupper ……267
12.10 wctype_t、wctype、iswctype ……268
12.11 wctrans_t、wctrans ……269
第13章 字符串处理 ……270
13.1 strcat、strncat、wcscat、wcsncat ……270
13.2 strcmp、strncmp、wcscmp、wcsncmp ……271
13.3 strcpy、strncpy、wcscpy、cscncpy ……272
13.4 strlen、wcslen ……273
13.5 strchr、strrchr、wcschr、wcsrchr ……273
13.6 strspn、strcspn、strpbrk、strrpbrk、wcsspn、wcscspn、wcspbrk ……274
13.7 strstr、strtok、wcsstr、wcstok ……275

13.8 strtod、strtof、strtold、strtol、strtoll、strtoul、strtoull ……276
13.9 atof、atoi、atol、atoll ……277
13.10 strcoll、strxfrm、wcscoll、wcsxfrm …277
第14章 内存函数 ……279
14.1 memchr、wmemchr ……279
14.2 memcmp、wmemcmp ……279
14.3 memcpy、memccpy、memmove、wmemcpy、wmemmove ……280
14.4 memset、wmemset ……281
第15章 输入/输出工具 ……282
15.1 FILE、EOF、wchar_t、WEOF ……283
15.2 fopen、fclose、fflush、freopen、fwide ……284
15.2.1 文件模式 ……285
15.2.2 文件的定向 ……286
15.3 setbuf、setvbuf ……287
15.4 stdin、stdout、stderr ……287
15.5 fseek、ftell、rewind、fgetpos、fsetpos ……288
15.5.1 fseek和ftell ……287
15.5.2 fgetpos和fsetpos ……290
15.6 fgetc、fgetwc、getc、getwc、getchar、getwchar、ungetc、ungetwc ……290
15.7 fgets、fgetws、gets ……291
15.8 fscanf、fwscanf、scanf、wscanf、sscanf、swscanf ……292
15.8.1 控制字符串 ……293
15.8.2 转换规范 ……293
15.9 fputc、fputwc、putc、putwc、putchar、putwchar ……298
15.10 fputs、fputws、puts ……299
15.11 fprintf、printf、sprintf、snprintf、fwprintf、wprintf、swprintf ……299
15.11.1 输出格式 ……300
15.11.2 转换规范 ……301
15.11.3 转换标志 ……301
15.11.4 最小字段宽度 ……302
15.11.5 精度 ……303
15.11.6 长度规范 ……303
15.11.7 转换操作 ……304
15.12 v[x]printf、v[x]scanf ……311
15.13 fread、fwrite ……312
15.14 feof、ferror、clearerr ……313
15.15 remove、rename ……314
15.16 tmpfile、tmpnam、mktemp ……314
第16章 基本工具 ……316
16.1 malloc、calloc、mlalloc、clalloc、free、cfree ……316
16.2 rand、srand、RAND_MAX ……318
16.3 atof、atoi、atol、atoll ……319
16.4 strtod、strtof、strtold、strtol、strtoll、strtoul、strtoull ……319
16.5 abort、atexit、exit、_Exit、EXIT_FAILURE、EXIT_SUCCESS …321
16.6 getenv ……322
16.7 system ……323
16.8 bsearch、qsort ……323
16.9 abs、labs、llabs、div、ldiv、lldiv ……325
16.10 mblen、mbtowc、wctomb ……326
16.10.1 编码和转换状态 ……327
16.10.2 长度函数 ……327
16.10.3 转换为宽字符 ……327
16.10.4 宽字符转换为多字节字符 ……328
16.11 mbstowcs、wcstombs ……328
16.11.1 转换为宽字符串 ……328
16.11.2 从宽字符串转换为多字节字符 …329
第17章 数学函数 ……331
17.1 abs、lbas、llabs、div、ldiv、lldiv ……331
17.2 fabs ……332
17.3 ceil、floor、lrint、llrint、lround、llround、nearbyint、round、rint、trunc ……332
17.4 fmod、remainder、remquo ……333
17.5 frexp、ldexp、modf、scalbn ……334
17.6 exp、exp2、expm1、ilogb、log、log10、log1p、log2、logb ……334

17.7 cbrt、fma、hypot、pow、sqrt ……335
17.8 rand、srand、RAND_MAX ……336
17.9 cos、sin、tan、cosh、sinh、tanh ……336
17.10 acos、asin、atan、atan2、acosh、asinh、atanh ……337
17.11 fdim、fmax、fmin ……338
17.12 类型通用的宏 ……338
17.13 erf、erfc、lgamma、tgamma ……341
17.14 fpclassify、isfinite、isinf、isnan、isnormal、signbit ……342
17.15 copysign、nan、nextafter、nexttoward ……343
17.16 isgreater、isgreaterequal、isless、islessequal、islessgreater、isunordered ……343
第18章 时间和日期函数 ……345
18.1 clock、clock_t、CLOCKS_PER_SEC、times ……345
18.2 time、time_t ……346
18.3 asctime、ctime ……346
18.4 gmtime、localtime、mktime ……347
18.5 difftime ……348
18.6 strtime、wcsftime ……349
第19章 控制函数 ……352
19.1 assert、NDEBUG ……352
19.2 system、exec ……352
19.3 exit、abort ……353
19.4 setjmp、longjmp、jmp_buf ……353
19.5 atexit ……354
19.6 signal、raise、gsignal、ssignal、psignal ……354
19.7 sleep、alarm ……356
第20章 区域 ……357
20.1 setlocale ……357
20.2 localeconv ……358
第21章 扩展整数类型 ……361
21.1 基本规则 ……361
21.1.1 类型的种类 ……361
21.1.2 要么都定义，要么都不定义 ……361
21.1.3 MIN和MAX限制 ……361
21.1.4 PRI...和SCN...格式字符串宏 ……362
21.2 固定长度的整数类型 ……363
21.3 具有最小宽度的最小长度类型 ……364
21.4 具有最小宽度的快速类型 ……364
21.5 指针长度和最大长度整数类型 ……365
21.6 ptrdiff_t、size_t、wchar_t、wint_t和sig_atomic_t的范围 ……366
21.7 imaxabs、imaxdiv、imaxdiv_t ……366
21.8 strtoimax、strtouimax ……367
21.9 wcstoimax、wcstoumax ……367
第22章 浮点环境 ……368
22.1 介绍 ……368
22.2 浮点环境 ……369
22.3 浮点异常 ……369
22.4 浮点四舍五入模式 ……370
22.5 浮点表达式收缩 ……371
第23章 复数运算 ……372
23.1 复数函数库约定 ……372
23.2 complex、_Complex_I、imaginary、_Imaginary_I、I ……372
23.3 CX_LIMITED_RANGE ……372
23.4 cacos、casin、catan、ccos、csin、ctan ……373
23.5 cacosh、casinh、catanh、ccosh、csinh、ctanh ……374
23.6 cexp、clog、cabs、cpow、csqrt ……374
23.7 carg、cimag、creal、conj、cproj ……375
第24章 宽字符和多字节字符工具 ……377
24.1 基本类型和宏 ……377
24.2 宽字符和多字节字符之间的转换 ……377
24.3 宽字符串和多字节字符串之间的转换 ……379
24.4 转换为算术类型 ……380
24.5 输入和输出函数 ……380
24.6 字符串函数 ……381
24.7 日期和时间转换 ……382
24.8 宽字符分类和映射函数 ……382
附录A ASCII字符集 ……383
附录B 语法 ……384
附录C 练习答案 ……396

第一部分 C 语 言



第1章 概述

第2章 词法元素

第3章 C预处理器

第4章 声明

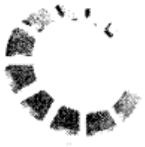
第5章 类型

第6章 转换和表示形式

第7章 表达式

第8章 语句

第9章 函数



第1章 概　述

20世纪70年代初，Dennis Ritchie在贝尔实验室设计了C语言。C语言的前身可以追溯到ALGOL（1960），历经剑桥的CPL（1963）、Martin Richards的BCPL（1967）以及Ken Thompson在贝尔实验室所开发的B语言（1970）发展而来。尽管C语言是一种通用用途的编程语言，但它在传统上用于系统编程。特别值得一提的是，著名的UNIX操作系统一开始就是用C语言编写的。

C语言流行的原因是多方面的。它小巧、高效，是一种功能强大的编程语言，并且具有丰富的运行时函数库。它提供了对计算机的精确控制，却没有采用太多的隐藏机制。由于C语言的标准化早在十多年前就已完成，所以程序员们已经习惯了使用C语言。一般来说，用C语言能够非常容易地编写可以在不同国家、不同语言的不同计算机系统中进行移植的程序。另外，现在存在大量的遗留C代码，它们需要修改和扩展。

从20世纪90年代后期开始，C语言的地位逐渐受到它的“大哥”C++的挤压。但是，C语言仍然有大量忠实的追随者，并且依然非常流行，因为程序员在很多时候并不需要使用C++所提供的特性，或者不喜欢C++所带来的其他开销。

C语言经受住了时间的考验，经验丰富的程序员可以用C语言编写出快速高效的代码，数以百万行计的C程序代码充分证明了它的生命力。

1.1 C语言的演变

1984年，当我们编写本书的第1版时，C语言已经广泛地使用。但是，它还没有制订官方标准，也不存在语言的精确描述。程序员们事实上所使用的标准是当时的C编译器。C语言在1989年成为一种国际标准，1994年进行了一次修订，1999年又进行了一次重要的修订。

简单地修改语言的定义并不会自动更改世界上数以百万行计的C程序代码。我们努力使本书与时俱进，使程序员在遇到各种变体的C语言时都能把本书作为参考手册。

1.1.1 传统C

C语言的最初描述出现在Brian Kernighan和Dennis Ritchie所著的《The C Programming Language》第1版（Prentice Hall，1978），通常称为“K&R”。在这本书出版之后，C语言又经历了一些较小的变化，添加了一些新特性，并删除了一些不合时宜的特性。我们把20世纪80年代早期公认的C语言定义称为传统C，也就是C语言被标准化之前的一种语言变体。当然，众多C语言编译器开发商都对传统C进行了各自的扩展。

1.1.2 标准C（1989）

1982年，美国国家标准委员会（ANSI）认识到对C语言进行标准化有助于C语言在商业编程

领域得到更广泛的使用，于是成立了一个委员会，着手制订C语言及其运行时函数库的标准。这个称为X3J11的委员会（现在称为NCTIS J11）由Jim Brodie担任主席，成功地制订了一个标准，并于1989年被采纳为《美国国家标准X3.159-1989》，或称ANSI C。

考虑到编程是一项国际性的活动，因此在完成了ANSIC标准之后，国际标准组织（ISO）又创立了一个称为ISO/IEC JTC1/SC22/WG14的国际标准化小组。这个小组由P.J.Plauger所领导，对ANSI C只进行了少量的编辑性修改，把它转换为国际标准ISO/IEC 9899:1990。随后，ANSI也采纳了ISO/IEC标准，人们把这种标准简单地称为“标准C”。由于C的标准最终仍然会被修改，所以我们把它称为标准C（1989），或简称为C89。

从传统C到C89的改变包括：

- 添加了真正的标准函数库。
- 新的预处理器命令和特性。
- 函数原型，允许程序员在函数声明中指定参数的类型。
- 增加了一些新的关键字，包括const、volatile和signed。
- 宽字符、宽字符串和多字节字符。
- 在转换规则、声明和类型检查方面的许多小改动和澄清。

1.1.3 标准C（1995）

作为对C语言标准的正常维护，WG14对C89进行了两处技术修订（缺陷修复）和一个补充（扩展）。总体来说，这次修订对C89的修改还是比较温和的，主要是增加了一些新的函数库，其结果就是我们所说的“C89修正案1”，或者“C95”。它对C89的修改包括：

- 3个新的标准库头文件：iso646.h、wctype.h和wchar.h。
- 几个新的标记和宏，用于替换有些国家的字符集中不存在的操作符和标点符号。
- printf/scanf函数家族的一些新的格式代码。
- 大量用于多字节字符和宽字符的新函数以及一些类型和常量。

1.1.4 标准C（1999）

ISO/IEC标准必须定期审查和更新。1995年，WG14开始对C语言标准进行更深入的修订，于1999年完成了这项任务并获得了批准。新标准称为ISO/IEC 9899:1999或“C99”，它代替了以前的标准（以及所有的修正和补充），现在已经成为C语言的官方标准。开发商们对它们的C语言编译器进行了更新，以遵循新标准。

C99在C89和C95的语言及函数库上添加了许多新特性，包括：

- 复数运算。
- 整数类型的扩展，包括更长的标准类型。
- 可变长度的数组。
- 布尔类型。
- 对非英语字符集提供了更好的支持。
- 对浮点类型提供了更好的支持，包括所有类型的数学函数。

• C++风格的注释（//）。

和C95相比，C99是一次相当大的修订，因为它不仅对语言本身作了一些修改，而且对标准函数库进行了扩展。C99标准文档的篇幅要明显多于C89标准文档。但是，这些修改是“按照C的精神”进行的，它们并没有改变C语言的本质。

1.1.5 标准C++

C++语言是在20世纪80年代早期由Bjarne Stroustrup在AT&T的贝尔实验室设计的。现在，它在很大程度上已经取代C语言成为主流的编程语言。大多数C语言编译器实际上是C/C++编译器，程序员可以根据需要选择这两种语言之一。C++语言本身也被标准化，即ISO/IEC 14882:1998，或“标准C++”。C++语言在C语言的基础上进行了很多扩展，满足了程序员开发大型程序的需要，包括增强的类型检查以及对面向对象编程的支持。但是，C++也是世界上最为复杂的编程语言之一，存在许多意想不到的陷阱。

标准C++近乎是（但不完全是）标准C的超集。由于C语言和C++语言标准是根据不同的方案开发的，因此它们之间可能存在一些不协调之处。而且，C语言有意与C++语言保持距离。例如，C语言并没有试图增加C++语言中“类”这个特性的简化版本。

用标准C和标准C++的公共子集编写C代码也是可行的，有些人把这种代码称为Clean C。用这种代码编写的程序既可以编译为C程序，也可以编译为C++程序。由于C++一般具有比标准C更严格的规则，因此Clean C可以看成是一个更好、更具移植性的子集。如果使用Clean C编写程序，必须考虑下面这些情况：

• Clean C程序必须使用函数原型。C++不再允许旧式风格的声明。

• Clean C程序必须避免使用C++关键字作为标识符，例如class和virtual。

另外还存在一些规则和区别，但它们通常不会造成问题。在本书中，我们解释如何编写能够被C++编译器所接受的标准C语言代码。我们并不讨论标准C中并不存在的那些C++特性（正是这些特性奠定了C和C++的区别）。

1.1.6 本书内容

本书描述了C语言的3种主要变型：传统C、C89和C99。书中说明了哪些特性是C89修正案1所增加的，并描述了C和和C++的公共子集Clean C。我们还将解释如何编写“良好”的C程序，即具有良好的可读性、可移植性和可维护性的程序。

官方的“标准C”是C99。但是，我们使用标准C这个术语表示从C89沿用到C99的那些特性和概念。我们会特意说明那些仅在C99中存在的特性，以便使用C89编译器的程序员可以避开这些特性。

1.2 应该使用哪种C语言

应该使用哪种C语言取决于读者所使用的是哪种C编译器，以及代码在移植性方面有什么要求。可供的选择有：

1. C99。这是标准C的当前版本。它具有所有最新的特性，但有些编译器可能还不支持它

(这个情况应该很快会得到改变)。

2. C89。这是标准C的前一个版本。大多数最近的C程序和大多数C编译器都是基于这个版本的C，包括它的修正案1。

3. 传统C。现在大多是为了维护旧的C程序而使用。

4. Clean C。它与C++保持兼容。

C99一般与C89保持向上兼容，而后者又与传统C保持向上兼容。遗憾的是，很难编写向下兼容的C代码。例如，以函数原型为例。它们在标准C中是可选的，在传统C中是禁止的，而在C++中则是必须的。幸运的是，我们可以使用C预处理器，根据具体使用的编译器版本（甚至可以根据标准C是否包含了修正案1扩展）来更改代码。因此，C程序可以保持与所有版本的C语言保持兼容。我们将在第3章中介绍如何使用预处理器来实现这个目的。第3.9.1节提供了一个例子。

如果不存在编译器或现有C代码的限制，显然应该使用标准C作为基本的语言选择。标准C编译器现在几乎随处可见。自由软件基金会的GNU C（gcc）就是一种免费的标准C编译器（具有许多扩展）。

1.3 C编程概述

我们相信读者已经熟悉像C这样的高级编程语言，但是在这里对C编程进行简单的介绍，应该能够向读者提供一些帮助。

C程序由一个或多个源文件（或称翻译单元）组成，每个源文件可以包含整个C程序的一部分，一般是一定数量的外部函数。常用的声明通常放在头文件中，后者通过特殊的#include命令（第3.4节）被包含到源文件中。C程序必须具有一个称为main（第9.9节）的外部函数，这个函数就是C程序的起点。

C编译器独立地处理每个源文件，并把C程序文本翻译为计算机可以理解的指令。编译器“理解”C程序，并分析它的正确性。如果程序中出现了编译器可以检测到的错误，编译器就会产生错误信息。否则，编译器就会产生通常称为目标文件或目标模块的输出。

当所有的源文件都被编译后，所有的目标模块便被提交给一个称为链接器的程序。链接器将会解释这些模块之间的引用，从标准运行时函数库添加函数，并检测一些程序错误，例如无法找到一个必需的定义等。链接器一般并不是C特定的，每个计算机系统具有一个标准链接器，可以用于由许多不同语言所编写的程序。链接器产生一个可执行程序，后者可以被调用或运行。尽管大多数计算机系统都会经历这些步骤，但程序员所看到的步骤可能有所不同。在诸如Microsoft的Visual Studio这样的集成开发环境中，这些过程在程序员面前被完全隐藏。在本书中，我们并不关注生成C程序的细节，读者应该查阅自己的计算机系统和编程文档。

例子

假设有一个称为aprogram的程序，它由两个C源文件hello.c和startup.c组成。hello.c文件可能包含下面这几行代码：

```
#include <stdio.h> /* defines printf */
void hello(void)
{
    printf("Hello!\n");
}
```

由于hello.c包含了将被这个程序的其他部分所使用的工具（hello函数），我们创建了一个头文件hello.h声明这些工具。它包含了下面这行：

```
extern void hello(void);
```

startup.c文件包含了main函数，它简单地调用hello函数：

```
#include "hello.h"
int main(void)
{
    hello();
    return 0;
}
```

在UNIX系统中，程序的编译、链接和执行是分两个步骤执行的：

```
% cc -o aprogram hello.c startup.c
% aprogram
```

第1行编译并链接两个源文件、添加所需的标准库函数、并把可执行程序写入到aprogram中。第2行执行这个程序。这个程序的输出是：

```
Hello!
```

其他非UNIX系统可能使用不同的命令。越来越多的现代编程环境向程序员提供了集成的图形界面。在这种环境中生成C应用程序只需要在菜单中选择一个命令或者点击一个按钮。

1.4 一 致 性

C程序和C编译器都可以遵循标准C。如果一个C程序只使用标准C所描述的语言和函数库所提供的特性，它就是严格遵循标准C的。这种程序的行为必须不能依赖C语言中被标准认为是未指定、未定义或由编译器定义的特性。Perennial公司和Plum Hall公司提供了一些标准C测试套件，可以帮助确定产品与标准的一致性。

遵循标准的C实现分为两种：宿主式和独立式。如果一个C实现接受所有严格遵循标准的程序，它就是一种遵循标准的宿主式实现。如果一个C实现接受所有严格遵循标准并且不使用除了float.h、iso646.h（C95）、limits.h、stdarg.h、stdbool.h（C99）、stddef.h和stdint.h（C99）之外的任何库函数的程序，它就是一种遵循标准的独立式实现。第10章列出了这些头文件的内容。独立式一致性是为了使C的实现适用于嵌入式系统或其他具有最小运行时支持的目标环境，例如不支持文件系统的环境。

一致性程序就是可以被一致性编译器所接受的程序。因此，一致性程序可以依赖于一致性C实现中一些非移植性的、由编译器所定义的特性。严格一致性程序则无法依赖这些特性（因此具有最大限度的可移植性）。

一致性C实现可以提供一些不改变任何严格一致性程序的含义的扩展。这就允许C实现添加库函数以及定义自己的#pragma指令，但不能引入新的保留标识符或改变标准库函数的操作。

编译器厂商不断向提供新的非一致性扩展，他们的客户对此也已习以为常。编译器可以使用开关启用（或禁用）这些扩展。

1.5 语法约定

本书用一种统一的风格表示C程序的格式。当我们指定一种C语言文法时，终端符号用等宽字体显示，就像在程序中所出现的那样：

```
表达式　参数列表　声明器
```

语法定义是由非终端符号的名称以及后面的一个冒号引入的。如果它具有几种候选表示形式表示，则分在不同的行显示：

字符：

```
打印字符
转义字符
```

当冒号后面有“下列之一”这句话时，它表示下面一行或多行中的每个终端符号都可以成为候选定义：

数字：下列之一

```
0　1　2　3　4　5　6　7　8　9
```

定义的可选成分用终端或非终端符号的后缀opt表示：

枚举常量定义：

```
枚举常量　枚举初始化值opt
```

初始化值：

```
表达式
{ 初始化值列表，opt }
```

第2章
词法元素

本章描述C语言的词法结构，即C源文件中可以出现的字符以及它们是如何形成词法单元（或称为标记）的。

2.1 字 符 集

C源文件就是从一个字符集中所选择的字符序列。C程序是使用下面这些字符编写的，这些字符是在ISO/IEC 10646的基本拉丁文（Basic Latin）部分定义的：

1. 52个大写和小写拉丁字母：

```
A B C D E F G H I J K L M N O P Q R S T
U V W X Y Z a b c d e f g h i j k l m
n o p q r s t u v w x y z
```

2. 10个数字：

```
0 1 2 3 4 5 6 7 8 9
```

3. 空格。

4. 水平制表符（HT）、垂直制表符（VT）和换页符（FF）等格式控制符。

5. 29个图形字符以及它们的官方名称（如表2-1所示）。

另外，还必须有办法把源程序划分为多个代码行。这个任务可以由一个字符或一个字符序列来完成，或者使用源字符集之外的其他机制（例如记录尾指示符）来完成。

表2-1 图形字符

字符	名称	字符	名称	字符	名称
!	感叹号	+	加号	"	双引号
#	序号字符	=	等号	{	左花括号
%	百分号	~	波浪符	}	右花括号
^	折音符	[	左方括号	,	逗号
&	和号	]	右方括号	.	句号、点号
*	星号	'	撇号、单引号	<	小于号
(	左括号	\|	竖杠	>	大于号
_	下划线	\	反斜杠	/	正斜杠
)	右括号	;	分号	?	问号
-	连字符、减号	:	冒号		

有些使用本国字符集的国家并没有包含表2-1中的所有图形字符。C89（修正案1）定义了三字符组（trigraph）和标记重拼（token respelling），允许用ISO 646—1083不变性代码集（Invariant Code Set）编写C程序。

C源程序有时候还使用其他字符，包括：

1. 像退格符（BS）和回车符（CR）这样的格式字符。

2. 其他的基本拉丁字符，包括$符号（美元符号）、@（商用的AT）和`（重音）。

格式字符被看成是空格，除此之外不会影响源程序。其他图形字符只可能出现在注释、字符常量、字符串常量以及文件名中。

参考：基本拉丁文 第2.9节；字符常量 第2.7.3节；注释 第2.2节；字符编码 第2.1.3节；字符转义码 第2.7.6节；执行字符集 第2.2.1节；字符串常量 第2.7.4节；标记重拼 第2.4节；三字符组 第2.1.4节。

2.1.1 执行字符集

在执行C程序时，字符集的解释方式并不一定和编写C程序时所使用的字符集相同。执行字符集中的字符是用源字符集中的对应字符表示的，或者用以反斜杠（\）开头的特殊的字符转义序列表示。

除了前面所提到的标准字符之外，执行字符集还必须包括：

1. null字符，它必须被编码为0这个值。

2. 换行符，表示行末标志。

3. 警告、退格和回车字符。

null字符用于标志字符串的结束；换行符用于在输入/输出时把字符流分割为不同的行（如果换行符事实上就存在于执行环境的文本流中，对程序员无疑很有吸引力。但是，运行时函数库的实现可以采用自己的方式对它进行模拟。例如，换行符可以被转换为输出中的记录尾指示符，而记录尾指示符也可以被转换为输入中的换行符）。

和源字符集一样，在执行字符集中包含格式字符如退格符、水平制表符、垂直制表符、换页符、回车符也很常见。另外，它还可以使用特殊的转义序列表示源程序中的这些字符。

当C程序在同一台计算机上编译和执行时，源字符集和执行字符集是相同的。但有时候C程序是跨机器编译的，也就是在一台计算机（宿主计算机）上编译，但在另一台计算机（目标计算机）上执行。当编译器计算一个涉及字符的常量表达式的编译时值时，它必须使用目标计算机的编码方式，而不是采用最自然（对于编译器编写者而言）的源字符集编码方式。

参考：字符常量 第2.7.3节；注释 第2.2节；字符编码 第2.1.3节；字符集 第2.1节；执行字符集 第2.2.1节；常量表达式 第7.11节；转义字符 第2.7.5节；文本流 第15章。

2.1.2 空白字符和行终止符

在C源程序中，空格、行末符、垂直制表符、换页符和水平制表符（如果存在）合称为空白字符（稍后讨论的注释也属于空白）。这些字符将被忽略，除非它们用于分隔相邻的标记，或者出现在字符常量、字符串常量和#include文件名中。空白字符可以用于设置C程序的布局，使它更容易阅读。

行末字符（或字符序列）标志着一行源程序的结束。在有些编译器中，回车符、换页符和垂直制表符等格式字符都可以终止源代码行，因此它们合称为行分隔符。为了确定预处理器所处理的行，行终止符是非常重要的。紧随行分隔符之后的那个字符被认为是下一行的第1个字符。如果第1个字符就是行分隔符，这一行就结束（表示空行），接下来以此类推。

为了把一个源代码行延续到下一行，可以在第1行的末尾使用反斜杠字符（\），或者使用标

准C的三字符组??/。这时，反斜杠和行末标志符将会被删除，并创建一个更长的逻辑源代码行。这个约定在预处理器命令以及字符串常量中总是有效的，这也是它最常用的场合，并且具有很好的移植性。标准C和许多非标准的编译器对它进行了扩展，使它可以作用于任何源程序行。从概念上说，源代码行的延续出现在预处理和C程序的词法分析之前，但在三字符组的处理以及任何多字节字符序列到源字符集的转换之后。

例子

在标准C中，甚至是标记也可以跨越多行。下面这两行

```
if (a==b) x=1; el\
se x=2;
```

相当于下面这一行代码

```
if (a==b) x=1; else x=2;
```

如果一个编译器把所有的非标准源字符都看成是空格或行分隔符，它分别应该像处理空白字符和行末标志字符一样处理它们。标准C建议编译器在第一次读取源程序时把所有这类字符转换为某种规范的表示形式，以便进行上述的处理。但是，程序员应该意识到依赖这种做法可能造成的结果。例如，可以预料到出现在反斜杠后面的换页符将会被去除。

大多数C编译器对行延续符之前和之后的源代码行的最大长度施加了限制。C89要求编译器允许逻辑源代码行的长度至少为509个字符；C99则要求至少允许4095个字符。

参考：字符常量 第2.7.3节；预处理器词法转换 第3.2节；源字符集 第2.1.1节；字符串常量 第2.7.4节；标记 第2.3节；三字符组 第2.1.4节。

2.1.3 字符编码

计算机的（执行）字符集中的每个字符都具有某种编码约定，也就是在计算机中具有一种数值表示形式。这种编码是非常重要的，因为C把字符转换为整数，这个整数的值就是该字符的编码约定。前面所列出的所有标准字符必须都具有唯一的、非负的整数编码形式。

一种常见的C编程错误就是以为计算机所使用的是某种编码方案，但实际上它所使用的是另一种编码方案。

例子

C表达式'z'-'A'+ 1计算Z和A的编码值之差加上1的结果，也就是字母表中的字符数量。事实上，在ASCII字符集编码方案中，这个结果是26。但是，在EBCDIC编码方案中，字母并不是连续编码的，这个计算的结果是41。

参考：源字符集和执行字符集 第2.1.1节。

2.1.4 三字符组

标准C增加了一组三字符组，使C程序可以只使用ISO 646-1083不变性字符集来编写。这个字符集是7位的ASCII码的一个子集，它在许多非英语国家的字符集中也极为常用。表2-2列出了由两个连续的问号字符所引入的三字符组。标准C还提供了一些标记的重拼（第2.4节）和<iso64.h>头文件，后者定义了一些可用于替换一些操作符的宏。但是，和三字符组不同，这些宏并不会被看成是字符串和字符常量。

表2-2 ISO三字符组

三字符组	替换字符	三字符组	替换字符
??(	[	??)	]
??<	{	??>	}
??/	\	??!	\|
??'	^	??-	~
??=	#		

在源程序中，三字符组的替换出现在词法分析（标记化）以及字符串和字符串常量中的转义字符被确认之前。C编译器一共认识9个三字符组，所有其他字符序列（例如??&）都不会被当成是三字符组而进行转换。另外，一个新的转义字符\?可以防止对类似三字符组的字符序列进行转换。

例子

如果需要在一个字符串中包含一个三字符序列，但这三个字符在正常情况下将被解释为一个三字符组，必须使用反斜杠转义字符引用这三个字符中的至少一个字符。因此，字符串常量“What?\?!”实际上所表示的是字符串常量“What??!”。

如果需要在一个字符串中包含一个反斜杠字符，必须使用连续2个反斜杠（第1个引用第2个）。另外，每个反斜杠可以转换为对应的三字符组。因此，字符串“??/??/”表示一个包含了单个\字符的字符串。

参考：字符集 第2.1节；转义字符 第2.7.5节；iso646.h 第11.9节；字符串连接 第2.7.4节；标记重拼 第2.4节。

2.1.5 多字节字符和宽字符

为了容纳可能包含大量字符的非英语字母，标准C引入了宽字符和宽字符串的概念。为了表示宽字符以及外部世界中面向字节的字符串，标准C还引入了多字节字符的概念。C89修正案1提供了扩展，以处理宽字符和多字节字符。

宽字符和宽字符串 宽字符是扩展字符集（extended character set）中的元素的二进制表示形式。它的类型是整数类型wchar_t，这个类型是在头文件stddef.h中声明的。C89修正案1添加了整数类型wint_t，它必须可以表示wchar_t类型的所有值，并且可以表示一个额外的、可以区分的非宽字符值，用WEOF表示。标准C并没有指定任何宽字符编码方案，但0这个值被保留为“空的宽字符值”。宽字符常量可以用一种特殊的常量语法来表示（第2.7.3节）。

例子

一般情况下，宽字符占据16位。因此，在32位的计算机上，wchar_t可以用short或unsigned short来表示。如果wchar_t用short表示，并且−1并不是合法的宽字符，则wint_t可以用short表示，WEOF可以用−1表示。但是，更典型的方法是用int或unsigned int表示wint_t。

如果编译器厂商选择不支持扩展字符集（美国的C编译器厂商经常采用这种做法），wchar_t可以被定义为char，“扩展字符集”就与常规的字符集相同。

宽字符串（wide string）是以一个null宽字符结尾的连续的宽字符序列。null宽字符就是表示形式为0的宽字符。除了null宽字符以及另外的WEOF之外，标准C并没有指定扩展字符集的编码

方案。宽字符常量可以用特殊的字符串常量来指定（第2.7.4节）。

多字节字符 在C程序中，宽字符可以作为一个单独的单元进行操作，但大多数外部媒介（例如文件）和C源程序是基于单个字节长度的字符的。熟悉扩展字符集的程序员设计了多字节编码（multibyte encoding），这是一种区域特定的方法，在字节长度的字符序列和宽字符序列之间进行映射。

多字节字符是源字符集或执行字符集中宽字符的表示形式（它们可能具有不同的编码形式）。因此，多字节字符串是常规的C字符串，但是这些字符可以被解释为一系列的多字节字符。多字节字符的形式以及多字节字符和宽字符之间的映射取决于定义的实现方式。在编译时，这种映射是针对宽字符常量和宽字符串常量执行的，标准函数库提供了一些函数，可以在运行时执行这种映射。

多节字字符可以使用依赖状态的编码（state-depedent encoding），即多字节字符的解释可能取决于前一个出现的多字节字符。一般情况下，这种编码利用了转移字符。这是一种控制字符，属于多字节字符的一部分，用于更改当前以及后续字符的解释。在一个多字节字符序列中，当前解释被称为编码的*转换状态*（conversion state），或称*转移状态*（shift state）。开始对一个多字节字符序列进行转换时，总是使用一个可区分的初始转换（转移）状态，并且这个状态经常在转换结束时被返回。

例子

A编码（我们在这个例子中所使用的一种假设编码）是一种取决于状态的编码方案，它具有两个转移状态："上"和"下"。↑字符把转移状态修改为"上"，↓字符把转移状态修改为"下"。"下"状态是初始状态，所有的非转移字符具有常规的解释。在"上"状态中，每个多字节字符由一对字母数字字符组成，它们以一种我们并未指定的方式定义了一个宽字符。

在下面这些字符序列中，每个序列包含了3个A编码形式的多字节字符，都从初始的转换状态开始。

```
abc    ab↑e3    ↑ab↓b↑23    ↓a↓b↓c
```

最后一个字符串包含了并非严格必需的转移字符。如果允许冗余的转移序列，多字节字符可以变成任意长度（例如↓↓…↓x）。除非知道转换状态位于多字节字符序列的开始，否则就无法对诸如abcdef这样的序列进行解析，它既可以表示3个也可以表示6位宽字符。

ab!?x这个序列在A编码形式下是非法的，因为在"上"转移状态中出现了非字母数字字符。a↑b这个序列是非法的，因为最后一个多字节字符过早结束。

多字节字符也可以使用状态无关编码（state-independent encoding），即一个多字节字符的解释并不依赖于前一个多字节字符（尽管可能需要在一个多字节序列中从头开始寻找一个在字符串中间开始的多字节字符）。例如，C的转义字符的语法（第2.7.5节）表示一种char类型的状态独立编码，因为反斜杠字符（\）改变了它后面的一个或多个字符的解释，形成一个char类型的值。

例子

B编码是另一种假设的编码方案，它是一种状态无关编码，并使用了一个特殊的字符，用▽表示，用于更改它后面的非null字符的含义。在B编码方案中，下面这些字符序列各自包含了3个多字节字符：

```
abc    ▽a▽b▽c    ▽▽▽▽▽▽    a ▽bc
```

在B编码方案中，▽▽▽这个序列是非法的，因为它的最后并不是一个非null字符。

标准C对多字节字符设置了一些限制：

1. 标准字符集中的所有字符都必须出现在编码中。

2. 在初始的转移状态中，标准字符集中的所有单字节字符都保留正常的解释，并不影响转换状态。

3. 包含全0的字符不管是什么转移状态都被认为是null字符。多字节字符不能使用全0的字节作为它的第二个或后续的字符。

总而言之，这些规则保证多字节序列能够作为正常的C字符串（即中间不会嵌入null字符）处理。并且，没有特殊多字节编码的C字符和多字节序列一样可以得到预期的解释。

多字节字符的源代码和执行用法 多字节字符可以出现在注释、标识符、预处理器头文件名、字符串常量和字符常量中。每个注释、标识符、预处理器头文件名、字符串常量和字符常量必须以初始转移状态开始并结束，并且必须由合法的多字节字符序列组成。源代码的物理表示形式中的多字节字符在任何词法分析、预处理或甚至延续行的连接之前就被确认。

例子

一个日文文本编辑程序允许在字符串常量和注释中使用日文字符。如果文本被写入到一个字节流文件中，则日文字符将被转换为多字节序列，这对于字符串常量而言是可以接受的，可以被标准C编辑器所理解。

在预处理期间，字符串和字符常量中的字符被转换为执行字符集，然后才被解释为多字节序列。因此，在形成多字节字符时，转义序列能使用到。在这个阶段之前，程序中的注释已经被删除，因此多字节注释中的转义序列可能没有什么意义。

例子

如果源字符集和执行字符集相同，并且如果'a'的值在执行字符集为141_8，那么字符串常量"▽aa"所包含的两个多字节字符与"▽\141\141"相同（B编码方案）。

参考：字符常量 第2.7.3节；注释 第2.2节；多字节转换工具 第11.7、11.8节；字符串常量 第2.7.4节；wchar_t、WEOF第11.1节；宽字符 2.7.3节；宽字符串 第2.7.4节；wint_t 第11.1节。

2.2 注 释

在标准C中，可以使用两种方法编写注释。传统的注释从两个字符/*开始，以接下来出现的第1对*/结束。注释可以包含任意数量的字符，并且总是被当成空白看待。

从C99开始，注释也可以从//字符开始，并延续到下一个行分隔符（但并不包括这个行分隔符）。这有可能会破坏旧式C程序的代码（不过可能性极小）。作为练习，读者可以判断在什么条件下会出现这种情况。

如果注释位于字符串或字符常量内部，或者位于其他注释的内部，它并不会被编译器当做注释。除了能够认识注释中的多字节字符和行分隔符之外，C编译器并不会检查注释的内容。

例子

下面这个程序包含了4个合法的C注释：

```
// Program to compute the squares of
// the first 10 integers
#include <stdio.h>
```

```
void Squares( /* no arguments */ )
{
    int i;
    /*
        Loop from 1 to 10,
        printing out the squares
    */
    for (i=1; i<=10; i++)
        printf("%d //squared// is %d\n",i,i*i);
}
```

在预处理之前，编译器会删除注释，因此注释内部的预处理器命令将无法被认识，注释内部的行分隔符也不会终止预处理器命令。

例子

下面这两个#define命令具有相同的效果：

```
#define ten (2*5)

#define ten /* ten:
               one greater than nine
            */ (2*5)
```

标准C规定，所有的注释被一个空格字符所代替，以便于对C程序作进一步的转换。但是，有些旧式的实现并不会插入任何空白字符。这将影响预处理器的行为，详情将在第3.3.9节讨论。

有一些非标准的C编译器实现了“可嵌套注释”，即注释中的每一对/*必须存在对应的*/。这并不是标准的实现，程序员不应该依赖这个特性。为了使程序保持兼容，注释的内部不应该包含/*字符。

例子

为了使编译器忽略一个C程序的很大一部分代码，最好使用预处理器命令封装这块代码：

```
#if 0
…
#endif
```

这个方法优于在文本前后插入/*和*/。如果采用后面这种方法，如果这段代码内部也存在注释，就不会担心出现/*-风格问题。

参考：#if预处理器命令 第3.5.1节；预处理器词法约定 第3.2节；空白 第2.1节。

2.3 标 记

根据本章剩余部分所讨论的规则，组成C程序的字符被收集为词法标记。标记一共分为5种类型：操作符、分隔符、标识符、关键字和常量。

当编译器按照从左向右的顺序收集字符时，它总是尽可能形成最长可能的标记，即使这样做所形成的并不是合法的C程序。相邻的标记可以用空白字符或注释进行分隔。为了防止混淆，标识符、关键字、整型常量、浮点型常量必须与后面的标识符、关键字、整型常量或浮点型常量分隔。

预处理器采用了略有不同的标记约定。具体地说，标准C预处理器把#和##看成是标记，这在传统C中是非法的。

例子

字符	C标记
forwhile	forwhile
b>x	b、>、x
b->x	b、->、x
b--x	b、--、x
b---x	b、--、-、x

在第4个例子中，b--x这个字符序列是非法的C语法。如果分解为b、-、-、x这些标记，虽然在语法上是合法的，但编译器并不允许进行这样的标记化。

参考：注释 第2.2节；常量 第2.7节；标识符 第2.5节；预处理器标记 第3.2节；关键字 第2.6节；标记合并 第3.3.9节；空白字符 第2.1节。

2.4 操作符和分隔符

表2-3列出了C标记中的操作符和分隔符（标点符号）。为了帮助程序员使用没有美国英语字符的I/O设备，C提供了替代性拼写<%、%>、<:、:>、%:和%:%，它们分别等于标点符号{、}、[、]、#、##。除了这些重拼之外，头文件iso64.h定义了对某些操作符进行扩展的宏。

在传统C中，复合赋值操作符被看成是两个独立的标记，也就是1个操作符和1个等号，可以用空格进行分隔。在标准C中，复合赋值操作符被看成是一个单独的标记。

参考：复合赋值操作符 第7.9.2节；iso64.h 第11.9节；预处理器标记 第3.2节；三字符组 第2.1.4节。

表2-3 操作符和分隔符

标记类型	标 记
简单操作符	! % ^ & * - + = ~ \| . < > / ?
复合赋值操作符	+= -= *= /= %= <<= >>= &= ^= \|=
其他复合操作符	-> ++ -- << >> <= >= == != && \|\|
分隔符	() [] { } , ; : ...
替代型标记重拼	<% %> <: :> %: %:%

2.5 标 识 符

标识符（或名称）由大小写拉丁字母、数字和下划线字符序列组成。标识符不能以数字开头，并且必须与现有的关键字不同。

从C99开始，标识符也可以包含统一字符名称（第2.9节）和其他因编译器而异的多字节字符。在使用统一字符名称时，不能把数字放在标识符的开始处，并且必须使用“类似字母”的字符，不能使用标点符号。C99标准（ISO/IEC 9899:1999，附录D）和ISO/IEC TR 10176-1998提供了一个准确的统一字符名称列表。

标识符：

标识符-非数字

标识符 标识符-非数字
标识符 数字

标识符-非数字:

非数字
统一字符名称
其他因编译器而异的字符

非数字：下列之一

```
A  B  C  D  E  F  G  H  I  J  K  L  M
N  O  P  Q  R  S  T  U  V  W  X  Y  Z
a  b  c  d  e  f  g  h  i  j  k  l  m
n  o  p  q  r  s  t  u  v  w  x  y  z
_
```

数字：下列之一

```
0  1  2  3  4  5  6  7  8  9
```

当两个标识符具有相同的拼写时（包括所有字母的大小写也必须相同），它们就是相同的。也就是说，标识符abc和aBc是不同的。

除了避免使标识符的拼写与关键字相同之外，C程序员还必须避免自己所使用的标识符意外与标准函数库中的名称相同，不论是当前的标准函数库还是语言标准所描述的“未来函数库指南”部分。标准C保留了所有以下划线开头并且后面跟一个大写字母或另一个下划线的标识符。程序员应该避免使用这类标识符。C编译器有时会使用这些标识符，对标准C进行扩展（或者由于其他目的）。

C89要求编译器至少能够区分标识符的前31个字符，C99把这个要求增加到63个。在考虑这个要求时，每个统一字符名称或多字节字符都被看成是1个字符。

例子

在标准C之前的编译器中，标识符的长度被限制在8个字符之内。因此，标识符countless和countlessone被认为是相同的标识符。一般而言，较长的名称可以提高程序的清晰性，减少发生错误的机会。使用下划线以及混合使用大小写字母可以提高较长标识符的可读性：

```
averylongidentifier
AVeryLongIdentifier
a_very_long_identifier
```

外部标识符（用存储类别extern声明的标识符）可能具有额外的拼写限制。这种标识符必须由其他软件（例如调试器或链接器）进行处理，在处理时可能会有更多的限制。C89所要求的最小容量是6个字符，并且不考虑大小写。C99把这个限制增加到31个字符，并区分大小写。但是，C99允许把统一字符名称当成是6个字符（最大为\U0000FFFF）或10个字符（\U00010000或更高）。即使在C99之前，大多数编译器允许外部名字至少为31个字符。

例子

当一个C编译器允许较长的外部标识符，但目标计算机却要求更短的外部名称时，可以使用预处理器隐藏这些较短的名称。在下面的代码中，一个外部错误处理函数具有较短（并因此比较难看明白）的名称eh73，但这个函数却用一个更具描述性的名称error_handler来表示。这是通

过声明一个展开结果为eh73的预处理宏error_handler实现的。

```
#define error_handler eh73
extern void error_handler();
…
error_handler("nil pointer error");
```

有些编译器允许在标识符中使用前面所指定字符的之外的其他字符。标识符中常常允许使用美元符号（$），使程序可以访问其他计算机系统所提供的特殊的非C库函数。

参考：#define命令 第3.3节；外部名称 第4.2.9节；关键字 第2.6节；多字节序列 第2.1.5节；保留的函数库标识符 第10.1.1节；统一字符名称 第2.9节。

2.6 关 键 字

表2-4所列出的标识符是标准C中的关键字，它们不能作为普通的标识符使用。它们可以作为宏名，因为所有的预处理都是在确认这些关键字之前进行的。关键字_Bool、_Complex、_Imaginary、inline和restrict是C99新增的。

表2-4 C99的关键字

auto	_Bool[a]	break	case
char	_Complex[a]	const	continue
default	restrict[a]	do	double
else	enum	extern	float
for	goto	if	_Imaginary[a]
inline[a]	int	long	register
return	short	signed	sizeof
static	struct	switch	typedef
union	unsigned	void	volatile
while			

a 这些关键字是C99新增的，在C++中并未保留。

除了表2-4所列出的关键字之外，标识符asm和fortran是常见的语言扩展。程序员还可能希望把头文件iso646.h所定义的宏（and、and_eq、bitand、bitor、compl、not、not_eq、or、or_eq、xor和xor_eq）作为保留标识符。这些标识符在C++中被保留。

例子

下面的代码是少数适合使用与关键字相同的名称作为宏名的例子之一。这个定义允许非标准编译器生成那些使用了void关键字的程序。

```
#ifndef __STDC__
#define void int
#endif
```

参考：_Bool 第5.1.5节；C++关键字 第2.8节；_Complex 第5.2.1节；#define命令 第3.3节；identifiers 第2.5节#ifndef命令 第3.5节；inline 第9.10节；<iso646.h>头文件 第11.5节；restrict 第4.4.6节；__STDC__ 第11.3节；void类型指定符 第5.9节；_Imaginary 第6.2.4节。

预定义标识符

C99引入了预定义标识符（predefined identifier）的概念，并定义了一个预定义标识符：__func__。它的性质和关键字相似，尽管它本身并不是关键字。和预定义宏不同，预定义标识符可以遵循正常的代码块作用域规则。和关键字一样，程序员无法自己定义预定义标识符。

如果下面这个声明出现在每个函数定义的左花括号之后，C99编译器就会隐式地声明标识符__func__：

```
static const char __func__[] = "function-name";
```

这个标识符可以由调试工具使用，打印出外层函数的名称，如下所示：

```
if (failed) printf("Function %s failed\n", __func__);
```

把C程序移植到内存限制较为严格的目标计算机时，如果程序在运行时并不需要这些字符串，C编译器必须仔细地将它们去除。

参考：函数定义 第9.1节；预定义宏 第3.3.4节；作用域 第4.2.1节。

2.7 常 量

常量的词法类型包括4种不同类型的常量：整数、浮点数、字符和字符串：

常量：

整数常量
浮点数常量
字符常量
字符串常量

这些标记在其他语言中称为字面值，以便与其他一些值为常量（即不会变化）但又不属于这些词法类型的对象进行区分。在C中，后面这类对象的一个例子是枚举常量，它属于标识符这个词法类型。在本节中，我们使用编译C的术语常量同时表示这两种情况。

每个常量都有值和类型。下面各节描述了不同类型的常量的格式。

参考：字符常量 第2.7.3节；枚举常量 第5.5节；浮点数常量 第2.7.2节；整数常量 第2.7.1节；字符串常量 第2.7.4节；标记 第2.3节；值 第7.3节。

2.7.1 整数常量

整数常量可以用十进制、八进制或十六进制记法来指定：

整数常量：

十进制常量 整数后缀$_{opt}$
八进制常量 整数后缀$_{opt}$
十六进制常量 整数后缀$_{opt}$

十进制常量：

非零数字
十进制常量数字

八进制常量：

0

```
八进制常量  八进制数字
```

十六进制常量：

```
0x 十六进制数字
0X 十六进制数字
十六进制常量 十六进制数字
```

数字：下列之一

```
0  1  2  3  4  5  6  7  8  9
```

非零数字：下列之一

```
1  2  3  4  5  6  7  8  9
```

八进制数字：下列之一

```
0  1  2  3  4  5  6  7
```

十六进制数字：下列之一

```
0  1  2  3  4  5  6  7  8  9
A  B  C  D  F  F  a  b  c  d  e  f
```

整数后缀：

```
long后缀  unsigned后缀opt
long long后缀  unsigned后缀opt                    (C99)
unsigned后缀  long后缀opt
unsigned后缀  long long后缀opt                    (C99)
```

long后缀：下列之一

```
l  L
```

long long后缀:下列之一

```
ll  LL
```

unsigned后缀：下列之一

```
u  U
```

下面是确定整数常量的基数的规则：

1. 如果一个整数常量以字母0x或0X开始，那么它就是十六进制常量，其中a-f（或A-F）分别表示10-15。

2. 否则，如果它以数字0开始，那么它就是八进制常量。

3. 否则，它就是十进制常量。

整数常量后面可以紧随后缀字母，指定了它的类型的最小长度：

- 字母l或L表示long类型的常量
- 字母ll或LL表示long long类型的常量（C99）
- 字母u或U表示unsigned类型（int、long或long long）的常量

unsigned后缀可以按照任何顺序与long或long long后缀组合使用。小写字母l很容易与数字1混淆，因此应该避免作为后缀使用。

如果没有溢出，整数常量的值总是非负的。如果它的前面有个负号，它是应用于这个常量的单目操作符，它本身并不是常量的一部分。

整数常量的实际类型取决于它的长度、基数、后缀字母，类型表示形式取决于C编译器。确定整数常量类型的规则非常复杂，并且在非标准C、C89和C99中是不相同的。表2-5显示了所有的规则。

如果某种类型的整数常量的值超出了表2-5中它自己那组中最后一个类型的最大可表示整数，其结果是未定义的。在C99中，编译器可以把这些大常量赋值为一种扩展整数类型，然后再加上表中的符号约定（如果所有的标准选择都是有符号的，则扩展类型也必须是有符号的。如果所有的标准选择都是无符号的，则扩展类型也必须是无符号的。在其他情况下，有符号和无符号都是可以接受的）。在C89中，和整数类型的表示形式有关的信息是由头文件limits.h提供的。在C99中，头文件stdint.h和inttypes.h包含了额外的信息。

为了说明整数常量的一些微妙之处，假设int类型使用16位的补码表示形式，long类型使用32位的补码表示形式，long long类型使用64位的补码表示形式。我们在表2-6中列出了一些有趣的整数常量、它们的真实数学值、它们的类型（在常规情况下以及在标准C规则下），以及用于存储常量的实际C表示形式。

在这张表中，一个值得注意的有趣之处是2^{15}和$2^{16}-1$之间的整数在写成十进制常量时为正值，但在写成八进制或十六进制常量时（被转换为int类型）为负值。尽管存在这种异常的情况，但程序员很少受到整数常量值的困扰，因为常量的表示形式即使在出现问题时仍然是相同的。

C99通过INTN_C、UINTN_C和UINTMAX_C这些宏对整数常量的长度和类型实行可移植的控制。这些宏是在stdint.h中定义的。

表2-5 整数常量的类型

常量	最初的C[a]	C89[a]	C99[a,b]
dd...d	int	int	int
	long	long	long
		unsigned long	long long
0dd...d	unsigned	int	int
0Xdd...d	long	unsigned	unsigned
		long	long
		unsigned long	unsigned long
			long long
			unsigned long long
dd...dU	无应用	unsigned	unsigned int
0dd...dU		unsinged long	unsigned long
0Xdd...dU			unsigned long long
dd...dL	long	long	long
		unsigned long	long long
0dd...dL	long	long	long
0Xdd...dL		unsigned long	unsigned long
			long long
			unsigned long long
dd...dUL	无应用	unsigned long	unsigned long
0dd...dL			unsigned long long
0Xdd...dL			
dd...dLL	无应用	无应用	long long
0dd...dLL	无应用	无应用	long long
0Xdd...dLL			unsigned long long

（续）

常量	最初的C[a]	C89[b]	C99[a,b]
dd...dULL 0dd...dULL 0xdd...dULL	无应用	无应用	unsigned long long

a 所选择的类型是对应的组中能够不溢出地表示常量值的第1个类型。

b 如果所列出的类型都不够大，并且存在扩展类型，就可以使用这种扩展类型。

表2-6 整数常量的类型赋值

C常量记法	真实值	传统类型	标准C类型	实际表示形式
0	0	int	int	0
32767	$2^{15}-1$	int	int	0x7FFF
077777	$2^{15}-1$	unsigned	int	0x7FFF
32768	2^{15}	long	long	0x00008000
010000	2^{15}	unsigned	unsigned	0x8000
65535	$2^{16}-1$	long	long	0x0000FFFF
0xFFFF	$2^{16}-1$	unsigned	unsigned	0xFFFF
65536	2^{16}	long	long	0x00010000
0x10000	2^{16}	long	long	0x00010000
2147483647	$2^{31}-1$	long	long	0x7FFFFFFF
0x7FFFFFFF	$2^{31}-1$	long	long	0x7FFFFFFF
2147483648	2^{31}	long[a]	unsigned long	0x80000000
			C99:long long	
0x80000000	2^{31}	long[a]	unsigned long	0x80000000
4294967295	$2^{32}-1$	long[a]	unsigned long	0xFFFFFFFF
			C99:long long	0x00000000FFFFFFFF
0xFFFFFFFF	$2^{32}-1$	long[a]	unsigned long	0xFFFFFFFF
4294967296	2^{32}	未定义	未定义	0x0
			C99：long long	0x0000000100000000
0x100000000	2^{32}	未定义	未定义	0x0
			C99：long long	0x0000000100000000

a 该类型无法准确地表示值。

例子

如果long类型采用32位的补码表示形式，下面这个程序可以判断实际使用的规则：

```
#define K 0xFFFFFFFF /* -1 in 32-bit, 2's compl. */
#include <stdio.h>
int main()
{
    if (0<K) printf("K is unsigned (Standard C)\n");
    else printf("K is signed (traditional C)\n");
    return 0;
}
```

参考：整数类型的转换 第6.2.3节；扩展整数类型 第5.1.4节；整数类型 第5.1节；

INTMAX_C 第21.5节；INTN_C 第21.3节；limits.h 第5.1.1节；溢出 第7.2.2节；stdint.h 第21章；单目负号操作符 第7.5.3节；无符号整数 第5.1.2节。

2.7.2 浮点数常量

浮点常量可以写成带小数点的形式，也可以写成带符号的指数形式，也可以两者皆采用。标准C允许使用一个后缀字母（浮点后缀）指定float和long double类型的常量。如果不使用后缀，浮点常量的类型默认为double：

浮点常量：

十进制浮点常量
十六进制浮点常量 （C99）

十进制浮点常量：

数字序列 指数 $浮点后缀_{opt}$
带小数点的数字 $指数_{opt}$ $浮点后缀_{opt}$

数字序列：

数字
数字序列 数字

带小数点的数字：

数字序列 .
数字序列 . 数字序列
. 数字序列

数字：下列之一

```
0  1  2  3  4  5  6  7  8  9
```

指数：

e $符号部分_{opt}$ 数字序列
E $符号部分_{opt}$ 数字序列

符号部分：下列之一

```
+  -
```

浮点后缀：下列之一

```
f  F  l  L
```

如果没有溢出，浮点常量的值总是非负的。如果浮点常量的前面有个负号，它是作用于这个常量的单目操作符，它本身并不是浮点常量的一部分。如果无法准确地表示一个浮点常量，编译器可以选择最接近的表示值V，或者略大小或略小于V的接近值。如果浮点常量的数值由于太大或太小而难以表示，其结果就是不可预测的。有些编译器会向程序员警告这个问题，但大多数编译器会悄然地换用其他可表示的值。在标准C中，浮点限制是在float.h头文件中指定的。像无限和NaN（不是数）这样的特殊浮点常量是在math.h中定义的。

在C99中，复数浮点常量被写成带有虚数常量_Complext_I（或I）的浮点常量表达式。虚数常量是在complex.h中定义的。

例子

下面这些是合法的十进制浮点常量：0.、3e1、3.14159、.0、1.0E-3、1e-3、1.0、.00034、

2e+9。下面这些浮点常量在标准C中是合法的：1.0f、1.0e67L、0E1L。

C99复数常量的一个例子是1.0 + 1.0 * I（如果已经包含了complex.h）。

C99允许用十六进制记法表示浮点常量。以前版本的C只允许十进制浮点常量。十六进制格式使用字母p分隔小数部分和指数部分，这是因为惯用的表示指数的字母e会与十六进制数字混淆。二进制指数是一种有符号十进制数，表示2的幂（并不是十进制常量所使用的10的幂，也不是有些人所猜想的16的幂）。

十六进制浮点常量： (C99)

十六进制前缀 带小数点的十六进制数字 二进制指数 浮点后缀$_{opt}$
十六进制前缀 十六进制数字序列 二进制指数 浮点后缀$_{opt}$

十六进制前缀：

0x
0X

带小数点的十六进制数字：

十六进制数字序列
十六进制数字序列 . 十六进制数字序列
. 十六进制数字序列

十六进制数字序列

十六进制数字
十六进制数字序列 十六进制数字

二进制指数

p 符号部分$_{opt}$ 数字序列
P 符号部分$_{opt}$ 数字序列

如果FLT_RADIX（float.h）不等于2，可能无法精确地表示十六进制浮点常量。如果无法精确地表示，它所指定的值必须是最接近的可表示值。

参考：complex.h 第23.2节；double类型 第5.2节；float.h 第5.2节；上溢和下溢 第7.2.2节；浮点类型的长度 第5.2节；单目负号操作符 第7.5.3节。

2.7.3 字符常量

字符常量写成一对单引号内的一个或多个字符。有一种特殊的转义机制可以表示那些不方便或无法在源程序中直接输入的字符值或数值。标准C允许在字符常量前面添加前缀L，表示宽字符常量。

字符常量：

'c字符序列'
L'c字符序列' (C99)

c字符序列：

c字符
c字符序列 c字符

c字符：

除了单引号（'）、反斜杠（\）和换行符之外的所有源字符

转义字符
统一字符名称 (C99)

可以通过使用转义字符在字符常量中包含单引号、反斜杠和换行符，如第2.7.5节所述。用转义字符来表示那些在源程序中难以阅读的字符（例如格式字符）是种很好的做法。C99还允许在字符常量中使用统一字符名称（第2.9节）。

如果字符常量前面没有前缀L，它的类型就是int。这种字符常量一般用于表示一个单独的字符或转义码（第2.7.7节），常量的值是执行字符集中对应的字符的整数编码。如果一个字符常量是从一个char类型的对象转换而来，就会计算它的整数值。例如，如果char类型是一种8位的有符号类型，字符常量'\377'就具有符号位，它的值是−1。如果符合下列条件，字符常量的值是由编译器所定义：

1. 在执行字符集中不存在对应的字符。
2. 在常量中出现了超过1个的执行字符。
3. 数值转义符的值在执行字符集中无法表示。

例子

下面是一些单字符常量的例子，以及它们在ASCII编码中的十进制值。

字符	值	字符	值
'a'	97	'A'	65
' '	32	'?'	63
'\r'	13	'\0'	0
'"'	34	'\377'	255
'%'	37	'\23'	19
'8'	56	'\\'	92

标准C的宽字符常量由前缀字母L所指定，它的类型是wchar_t，这是stddef.h头文件所定义的一种整数类型。宽字符常量的目的是允许C程序员表示那些无法用char类型表示的字符（例如日文字符）。宽字符常量一般由一系列的字符和转义码组成，一起形成了一个单独的多字节字符。多字节字符与对应的宽字符之间的映射是由编译器定义的，对应于mbtowc函数，这个函数在运行时执行多字节字符到宽字符的转换。如果多字节字符使用了转移状态编码，则宽字符常量必须以初始转移状态开始和结束。如果一个宽字符常量包含了超过1个的宽字符，它的值是由编译器所定义的。

多字符常量 整数常量和宽字符常量可以包含一系列的字符。把这个字符序列映射到执行字符集之后，仍然可能存在超过1个的执行字符。这种常量的含义是由编译器决定的。

旧式编译器所使用的一种约定是把4字节的整数常量表示了为一个4字符常量，例如'gR8t'。这种用法是不可移植的，因为有些计算机可能不允许这种做法，而且不同的计算机的整数类型的长度以及“字符顺序”（即字符包装为字的顺序）可能不同。

例子

在一种4字节整数并且字节从左向右包装的ASCII实现中，'ABCD'的值将是41424344_{16}（'A'的值是0x42，'B'的值是0x42，接下来以此类推）。但是，如果使用了从右向左的包装，'ABCD'的值将是44434241_{16}。

参考：ASCII字符 附录A；字节顺序 第6.1.2节；字符码 第2.1节；char类型 第5.1.3节；转义字符 第2.7.5节；格式字符 第2.1节；mbtowc工具函数 第11.7节；多字节字符 第2.1.5节；wchar_t 第11.1节。

2.7.4 字符串常量

字符串常量是位于一对双引号内部的字符序列（可以为空）。程序员可以使用和字符常量相同的转义机制表示字符串中的字符。标准C允许在字符串常量前面添加前缀L，指定宽字符串常量。

字符串常量：

" s字符序列$_{opt}$ "
L " s字符序列$_{opt}$ " (C89)

s字符序列：

s字符
s字符序列 s字符

s字符：

除了双引号（"）、反斜杠（\）和换行符之外的所有源字符
转义字符
统一字符名称 (C99)

双引号、反斜杠和换行符可以通过使用转义字符包含在字符串常量中，如第2.7.5节所述。用转义字符来表示那些在源程序中难以阅读的字符（例如格式字符）是种很好的做法。C99允许在字符串常量中使用统一字符名称（第2.9节）。

例子

下面列出了5个字符串常量。

```
""
"\""
"Total expenditures: "
"Copyright 2000 \
Texas Instruments. "
"Comments begin with '/*'.\n"
```

第4个字符串相当于"Copyright 2000 Texas Instruments."，在0和T之间并没有换行符。

对于包含了n个字符的非宽字符串，在运行时将会静态分配一块可以容纳n+1个字符的内存，其中前n个字符是该字符串中的字符，最后一个字符是null字符'\0'。这块内存相当于这个字符串常量的值，它的类型是char [n+1]。类似，宽字符串常量由n个宽字符以及后面的一个null宽字符组成，它的类型是wchar_t [n +1]。

例子

sizeof操作符返回它的操作数的长度，而strlen函数（第13.4节）返回一个字符串的字符数量。因此，sizeof("abcdef")是7而不是6，sizeof("")是1而不是0，strlen("abcdef")是6，strlen("")是0。

如果一个字符串常量不是&操作符的参数，也不是sizeof操作符的参数，也不是作为字符数组的初始化值，就可以应用寻常数组转换，把这个字符串转换为由一个指针所指向的字符数组（指针指向字符串中的第1个字符）。

例子

char *p = "abcdef" 这个声明产生一个指针p，它被初始化为指向一块存储了7个字符的内存。

这7个字符分别是'a'、'b'、'c'、'd'、'e'、'f'和'\0'。

单字符的字符串常量的值和字符常量的值是不同的。int X = (int) "A" ；这个声明使X被初始化为一个指向存储了2个字符的内存块（包含'A'和'\0'）的指针的整数值（前提是这个指针可以用int类型来表示）。但是，int Y = (int) 'A' ；这个声明使Y被初始化'A'的字符码（在ISO 646编码中为0x41）。

字符串常量的存储 存储了字符串常量的字符的内存是无法修改的，因为它是只读的，也就是存在对修改的物理保护。有些函数（例如mktemp）要求向它传递指向字符串的指针，以便对字符串进行修改。因此，不要向这类函数传递字符串常量。反之，初始化一个包含了字符串常量的所有字符的字符数组（非const），并把这个数组的第1个元素的地址传递给这类函数。

例子

考虑下面这3个声明：

```
char p1[]= "Always writable";
char *p2 = "Possibly not writable";
const char p3[] = "Never writable"; /* Standard C only */
```

p1、p2和p3的值都是指向字符数组的指针，但它们在写入能力方面各不相同。p1[0] = 'x'这个赋值总是可行。p2[0] = 'x'可能成功，也可能导致一个运行时错误。p3[0] = 'x'总是会导致一个编译错误，这是由于const的含义所致。

不要以为所有的字符串常量会存储在不同的地址。标准C允许编译器为两个包含相同字符的字符串常量使用相同的存储地址。

例子

下面是一个简单的程序，显示了各种不同的字符串实现的区别。如果字符串常量是在只读内存中分配的，对string1[0]的赋值将会导致运行时错误。

```
char *string1, *string2;
int main() {
   string1 = "abcd"; string2 = "abcd";
   if (string1==string2) printf("Strings are shared.\n");
   else printf("Strings are not shared.\n");
   string1[0] = '1'; /* RUN-TIME ERROR POSSIBLE */
   if (*string1=='1') printf("Strings writable\n");
   else printf("Strings are not writable\n");
   return 0;
}
```

字符串的延续 一个字符串常量一般出现在源程序文本的一行中。如果字符串太长，不方便写在一行中，可以在包含这个字符串的所有文本行（除最后一行外）后面加上一个反斜杠字符\。在这种情况下，反斜杠字符和行末符号将被忽略。这就允许把一个字符串常量分在多行书写。有些旧式的编译器会删除延续行的前导空白字符，但这是不正确的做法。

标准C会自动连接相邻的字符串常量和相邻的宽字符串常量，并在最后一个字符串的末尾添加一个null字符。因此，在标准C中，\延续机制的一种替代方法就是把长字符串分隔为几个字符串。在C99中，宽字符串和常规的字符串也可以按照这种方式进行连接，产生一个宽字符串常量。在C89中，这种做法是不允许的。

例子

字符串s1的初始化在标准C和标准C之前的编译器中是允许的，但字符串s2的初始化则只被标准C所允许：

```
char s1[] = "This long string is acc\
eptable to all C compilers.";
char s2[] = "This long string is permissible "
            "in Standard C.";
```

换行符（即执行字符集中的行末符）可以通过在字符中常量中增加一个转义序列\n插入到这个字符串中。这应该不会与字符串常量所使用的行延续符混淆。

宽字符串 在标准C中，字符串常量前面如果有一个前缀字母L，它就是个宽字符串常量，它的类型是“wchar_t的数组”。它表示来自一个扩展执行字符集的宽字符序列，可能为诸如日语这样的语言所使用。宽字符串常量中的字符是多字节字符串，将按照一种取决于编译器的方式映射到一个宽字符序列（mbstowcs函数在运行时执行类似的任务）。如果多字符字符使用了一种转换状态编码，这个宽字符串常量必须以起始转移状态开始和结束。

参考：数组类型 第5.4节；const类型指定符 第4.4.4节；数组类型的版本 第6.2.7节；转义字符 第2.7.5节；初始值 第4.6节；mbstowcs工具函数 第11.8节；mktemp工具函数 第15.16节；多字节字符 第2.1.5节；指针类型 第5.3节；预处理器词法约定 第3.2节；sizeof操作符 第7.5.2节；strlen工具函数 第13.4节；空白字符 第2.1节；普通单目转换 第6.3.3节；wchar_t 第11.1节；统一字符名称 第2.9节。

2.7.5 转义字符

转义字符可以在字符常量和字符串常量中表示难以或无法直接在源程序中表示的字符。转义字符可以分为两种类型，一种是“字符转义”，可以用于表示一些特殊的格式和特殊的字符，另一种是“数值转义”，允许用一个数值编码来指定一个字符。C99还增加了统一字符名称作为转义字符。

转义字符：

```
\ 转义码
统一字符名称                    (C99)
```

转义码

```
字符转义码
八进制转义码
十六进制转义码                  (C89)
```

字符转义码：下列之一

```
n   t   b   r   f
v   \   '   ''
a   ?                           (C89)
```

八进制转义码：

```
八进制数字
八进制数字   八进制数字
八进制数字   八进制数字   八进制数字
```

十六进制转义码：

x 十六进制数字
十六进制转义码 十六进制数字 (C89)

这些转义字符的含义将在下面各节中讨论。

如果反斜杠后面的字符既不是个八进制数字，也不是字母x，也不是前面所列出的转义码之一，其结果就是未定义的（在传统C中，这种反斜本将被忽略）。在标准C中，反斜杠后面的所有小写字母被保留，用于将来的语言扩展。大写字母可以用于由编译器所定义的特定扩展。

参考：统一字符名称 第2.9节。

2.7.6 字符转义码

字符转义码用于表示一些常用的特殊字符，并且与目标计算机的字符集无关。表2-7列出了可以出现在反斜杠后面的字符以及它们的含义。

表2-7 字符转义码

转义码	转换	转义码	转换
a[a]	警告（即响铃）	v	垂直制表符
b	退格	\	反斜杠
f	换页	'	单引号
n	换行	"	双引号
r	回车	?[a]	问号
t	水平制表符		

a 标准C所增加。

转义码\a一般映射到“响铃”或输出设备上的其他声音信号（例如ASCII的control-G，它的值是7）。转义码\?用于在较罕见的情况下表示问号（在这些情况下如果直接使用问号，可能会被误认为是三字符组的一部分）。

引用标记（"）可能出现在没有前导反斜杠的字符常量中，撇号（'）可能出现在没有反斜杠的字符串常量中。

例子

为了说明如何使用字符转义码，下面是一个小程序，对输入的行数（实际上是换行符的数量）进行计数。getchar函数返回下一个输入字符，直到到达了输入的结束，此时getchar返回EOF宏的值，后者是在stdio.h中定义的。

```
#include <stdio.h>
int main(void) /* Count the number of lines in the input. */
{
    int next_char;
    int num_lines = 0;
    while ((next_char = getchar()) != EOF)
        if (next_char == '\n')
            ++num_lines;
    printf("%d lines read.\n", num_lines);
    return 0;
}
```

参考：字符常量 第2.7.3节；EOF 第15.1节；getchar工具函数 第15.6节；stdio.h 第15.1节；

字符串常量 第2.7.4节；三字符组 第2.1.4节。

2.7.7 数值转义码

数值转义码允许直接用八进制（在标准C中，还允许十六进制）编码值表示执行字符集中的一个字符。它最多允许使用3个八进制数字以及任意数量的十六进制数字。但是，标准C禁止使用unsigned char范围之外的值表示正常的字符常量，并禁止使用wchar_t范围之外的值表示宽字符常量。例如，在ASCII编码中，字符'a'可以被写成'141'或'x61'，字符'?'可以写成'\77'或'x3F'。用于终止字符串的null字符总是写成\0。如果一个数值转义码的值并不对应于执行字符集中的某个字符，其结果由编译器决定。

例子

下面这段简短的代码显示了数值转义码的用法。变量inchar的类型为int。

```
for (;;) {
    inchar = receive();
    if (inchar == '\0') continue;          /* Ignore */
    if (inchar == '\004') break;           /* Quit */
    if (inchar == '\006') reply('\006'); /* ACK */
    else reply('\025');                    /* NAK */
    }
```

在使用数值转义码时，程序员必须小心，原因有二。首先，数值转义码可能依赖于字符编码，因此是不可移植的。用宏定义来隐藏转义码是一种更好的做法，这样可以方便地对它们进行修改：

```
#define NUL '\0'
#define EOT '\004'
#define ACK '\006'
#define NAK '\025'
```

其次，数值转义码的语法是脆弱的。当八进制数字达到3个或者遇到第1个字符不是八进制数字的字符时，八进制转义符就会终止。因此，字符串"\0111"由2个字符组成，分别是\011和1。字符串"\090"由3个字符组成，分别是\0、9和0。十六进制转义序列也会遇到这种终止问题，尤其是因为它们的长度可以是任意的。为了在一个字符串中停止一个标准C的十六进制转义码，可以分割字符串：

```
"\xabc"      /* This string contains one character. */
"\xab" "c"   /* This string contains two characters. */
```

有些非标准的C编译器所提供的十六进制转义序列和八进制转义序列一样，只允许固定数量的十六进制数字。

参考：字符常量 第2.7.3节；#define 第3.3节；宏定义 第3.3节；null字符 第2.1节；字符串常量 第2.7.4节；执行字符集 第2.1节。

2.8 C++兼容性

本节列出了C和C++的一些词法区别。

2.8.1 字符集

标准C的标记重拼和三字符组也是C++标准的一部分，但它们在标准C++之前的编译器中并不常见。C和C++都允许统一字符名称，它们所使用的语法相同，但只有C明确允许在标识符中使用其他由编译器所定义的字符（个人希望C++编译器以扩展的形式提供这个功能）。

2.8.2 注释

C99的注释可以为C++所接受，反之亦然。在C99之前，标准C并不使用//作为注释起始符，因此字符序列//*在C和C++中的解释并不相同（它的细节作为练习留给读者）。

2.8.3 操作符

C++增加了3个新的复合操作符：

```
.*    ->*    ::
```

由于这些标记的组合在标准C程序中是非法的，所以不会影响C程序移植为C++程序。

2.8.4 标识符和关键字

表2-8所列出的标识符是C++的关键字，但不是C的关键字。但是，标准C保留了关键字wchar_t，C99还保留了关键字bool、true、false，作为标准函数库的一部分。

表2-8 额外的C++关键字

asm	export	private	throw
bool	false	protected	true
catch	friend	public	try
class	mutable	reinterpret_cast	typeid
const_cast	namespace	static_cast	typename
delete	new	template	using
dynamic_cast	operator	this	virtual
explict			wchar_t

2.8.5 字符常量

在C中，单字符常量的类型是int，但在C++中为char。多字符常量（它们依赖于编译器）在两种语言中都是int类型。在实际使用中，这点差别很少会产生问题，因为在整数上下文环境中所使用的C++字符常量会被隐式地提升为int类型。但是，sizeof('c')在C++中等于sizeof(char)，但在C中等于sizeof(int)。

2.9 关于字符集、指令集和编码

在C语言所产生的时代，人们尚未理解国际性的、多语言编程社区的需要。标准C对C语言进行了扩展，以容纳这个社区。本节简单介绍了标准C所解决的一些问题（以及相关的历史），使C语言对于非英语用户更为友好。

指令集和ASCII 每一种文化都根据一种可打印字母或符号的字符指令集进行书面交流。对于美国英语而言，这个指令集包括通常的52个大写和小写字母、十进制数字以及一些标点符号。

这些字符一共大约100个，它们（由美国英语程序员和计算机生产厂商）通过一种7位的编码（称为ASCII）被赋予特定的二进制值。经过编码的字符出现在标准键盘上，并可以在C语言定义中找到。

遗憾的是，其他文化具有不同的指令集。例如，英国的英语使用者所使用的是£而不是$，但7位的ASCII码并没有包含这个字符。像俄语和希伯来语这样的语言使用和英语完全不同的字母，像汉语、日语和朝鲜语（CJK）这样的语言所使用的指令集由几千个符号组成。当今的程序员常常需要创建要求读者和写入多种语言的文本，包括他们的母语。他们还需要在程序中使用母语注释和母语变量名。他们所编写的程序需要移植到其他文化，至少在其他文件中不会成为非法（除非读者理解梵语并且读者的计算机可以显示梵语字符，否则读者就无法读懂梵语注释）。

这个问题逐渐被人们所认识，人们设计了一些不完整的解决方案，这些解决方案至今仍然得到支持。例如，ISO 646-1083不变性代码集被定义为ASCII的一个子集，在许多非英语的字符集中较为常用。人们又发明了一些方法，替换在更小的字符集中不存在的C字符，例如{、}、[、]和#。

ISO/IEC 10646 字符集的通用解决方案是由ISO/IEC标准10646（加上修正案）定义的，即*通用多八位编码字符集*（Universal Multiple-Octet Coded Character Set，UCS）。它定义了一种4字节（或4个八位编码）的编码方案UCS-4，可以表示地球上所有文化的所有字符，并保留了充足的空间。UCS-4有一个非常有用的16位子集，称为基本多语言平面（Basic Multilingual Plane），即UCS-2，它由前两个字节为0的UCS-4编码所组成。UCS-2可以表示绝大多数文化的指令集，包括大约2万个CJK象形文字。但是，16位还不足以表示的有的字符，在计算机上操纵不低于32位的长度时显得不便，因此UCS-4是合适的方案。

Unicode字符集标准最初是由Unicode协会（www.unicode.org）所制订的一个16位的编码方案。Unicode 3.0现在完全与ISO/IEC 10646兼容。以前版本的Unicode只与UCS-2兼容。Unicode网站对字符编码提供了非常好的技术介绍。

字符集标准UCS-4、UCS-2和Unicode都与ASCII兼容。高8位为全0的只使用低8位的16位字符是一种扩展的ASCII字符，称为Latin-1。最初版本的7位ASCII字符现在称为Basic Latin，它是前9位均为0的UCS-2字符。

宽字符和多字节字符 字符表示形式大于传统的8位的字符称为宽字符。遗憾的是，8位（或7位）的字符并不容易根除。许多计算机和遗留的应用程序基于8位的字符，人们设计了计多方案，用8位或7位字符序列表示更大的字符指令集和宽字符。它们称为多字节编码或多字节字符。宽字符一般使用固定长度的表示形式，但多字节字符一般使用一个字节表示一些字节，使用两个字节表示另一些字符，使用三个字节表示其他一些字符，接下来以此类推。其中一个或多个8位字符被当成“转义”或“转移”字符，用于开始多字节序列。

我们今天在标准C中所看到的是一些技术的组合：有些方法用于处理明显的ASCII变型（三字符组和两字符组），有些方法用于处理完全现代的宽字符环境，有些方法用于处理I/O期间的多字节字符。最近，有一种方法可以按照一种可移植的方式表示任何文化适应的C程序（统一字符以及标识符中特定于本地文件的字符）。

统一字符名称 C99引入了一个概念，允许在字符常量、字符串常量和标识符中指定任何

UCS-2或UCS-4字符。它的语法如下：

统一字符名称：

\u 十六进制4位数
\U 十六进制4位数

十六进制4位数：

十六进制数字 十六进制数字 十六进制数字 十六进制数字

每个十六进制4位数包括4个十六进制数字，可以指定一个16位的值。十六进制4位数的值在ISO/IEC 10646中是以统一字符的4位数字和8位数字的“短标识符”指定的。\unnnn所指定的字符与\u0000nnnn所指定的字符相同。

C并不允许短标识符小于00A0的统一字符名称，但0024($)、0040(@)和0060(')例外。C也不允许短标识符在D800到DFFF范围内的统一字符名称。这些是控制字符，包括DELETE以及为UTF-16所保留的字符。使用标记合并创建统一字符名称的结果是未定义的。

参考：标识符和统一字符名称 第2.5节；标记合并 第3.3.9节。

2.10 练 习

1. 下面哪些是词法标记？
 (a) 关键字
 (b) 注释
 (c) 空格
 (d) 十六进制常量
 (e) 三字符组
 (f) 宽字符串常量
 (g) 括号
2. 假设下面这些由源字符组成的字符串是由C编译器处理的。哪些标记将被确认为一系列的C标记？每个字符串中可以找到几个标识符（不必担心有些标记序列可能无法出现在合法的C程序中）？
 (a) `X++Y`
 (b) `-12uL`
 (c) `1.37E+6L`
 (d) `"String""FOO"""`
 (e) `"String+ \"FOO\""`
 (f) `x**2`
 (g) `"X??/"`
 (h) `B$C`
 (i) `A*=B`
 (j) `while##DO`
3. 删除下面这个C程序片段中的所有注释。

```
/**/*/*"*/*/*"//*//**/*/
```

4. 一个标准C编译器必须在一个输入程序中执行下面每个操作。这些操作是按什么顺序执行的？
 把字符收集为标记
 删除注释
 转换三字符组
 处理行的延续
5. 下面显示了一些比较差的标识符，它们为什么比较差？

(a) `pipesendintake`
(b) `Const`
(c) `10`
(d) `O77U`
(e) `SYS#input`

6. 编写一些简单的标准C代码码，使它们分别因为下面的原因导致它们在C++中是非法的或者有不同的解释。
(a) C89不存在//风格的注释
(b) 常量类型
(c) 关键字冲突



第3章 C预处理器

C预处理器是一种简单的宏处理器。从概念上说，它在编译器读取源程序之前对C程序的源文本进行处理。在C的有些实现中，预处理器实际上是一个独立的程序。它读取最初的源文件，并写入到一个新的“经过预处理”的源文件，后者可以作为C编译器的输入。在C的其他实现中，由一个程序一次完成对源文件的预处理和编译。

3.1 预处理器命令

预处理器是由特殊的预处理器命令行控制的，它们是以#符号开头的源文件行。不包含预处理器命令的行称为源程序文本行。表3-1列出了预处理命令。

预处理器一般从源文件中删除所有的预处理器命令行，并在源文件中执行这些预处理命令所指定的转换操作。例如，对一个出现在源程序文本中的宏调用进行展开。经过预处理所产生的源代码文本必须仍然是一个合法的C程序。

预处理器代码行的语法与C语言其他部分的语法完全独立（尽管存在一些相似之处）。例如，把一个宏定义展开为一个语法上不完整的代码片段是完全可能的，只要这个片段在这个宏被调用时的所有上下文环境中都是合理的（也就是被正确地结束）。

表3-1 预处理器命令

命　令	含　义	参见章节
#define	定义一个预处理器宏	3.3
#undef	取消一个预处理器宏的定义	3.3.5
#include	插入另一个源文件的文本	3.4
#if	根据一个常量表示式的值，有条件地包含一些文本	3.5.1
#ifdef	根据一个宏名是否已被定义，有条件地包含一些文本	3.5.3
#ifndef	根据与#ifdef正好相反的测试，有条件地包含一些文本	3.5.3
#else	如果它前面的#if、#ifdef、#ifndef或#elif测试失败，就包含一些文本	3.5.1
#endif	终止条件文本	3.5.1
#line	提供用于编译器信息的行号	3.6
#elif[a]	如果它前面的#if、#ifdef、#ifndef或#elif测试失败，根据另一个常量表达式的值，包含一些文本	3.5.2
defined[a]	如果一个名称已经被定义为预处理器宏，这个预处理器函数返回1，否则返回0。它由#if和#elif所使用	3.5.5
#操作符[b]	用一个包含参数值的字符串常量替换一个宏参数	3.3.8
##操作符[b]	组合2个相邻的标记，创建单个标记	3.3.9
#pragma[b]	指定依赖编译器的信息	3.7
#error[b]	用指定的信息产生一个编译时错误	3.8

a 它原来并不是C的一部分，但现在在ISO和非ISO的C编译器中极为常见。

b 标准C新增的。

3.2 预处理器词法约定

预处理器并不会对源文本进行解析，但是会把它分解为标记，以便查找宏调用。预处理器的词法约定与编译器有点不一样。预处理器能够认识正常的C标记，另外还会把其他在C中并没有被认为是合法C标记的字符也看成为“标记”。因此，预处理器能够认识文件名、判断是否存在空格以及确定行尾标志符的位置。

以#开始的行被看成是预处理器命令，命令的名称必须紧随#字符之后。标准C允许在有些源代码行的#字符前后出现空格，但一些旧式的编译器不允许这种做法。如果一行中唯一的非空白字符就是#字符，它被标准C看成是null指令，被当成空行处理。有些旧式的编译器可能会采取不同的行为。

在预处理器命令行中，命令名称的后面可能包含了该命令的参数。如果一个预处理器命令不接受任何参数，则命令行的剩余部分应该是空，或者是空白字符或注释。许多ISO标准之前的编译器会无声无息地忽略预期的参数之后的所有字符（如果有）。这可能导致移植性问题。预处理器命令的参数一般受到宏替换的影响。

预处理器行是在宏扩展之前被认识的。因此，如果一个宏被展开之后是一些看上去像预处理器命令这样的东西，这个命令将不会被标准C编译器（以及大多数其他C编译器）所认识（有些老的UNIX系统违反了这个规则）。

例子

下面这些代码的效果是不在被编译的程序中包含math.h文件：

```
/* This example doesn't work as one might think! */
#define GETMATH #include <math.h>
GETMATH
```

展开后的标记序列

```
# include < math . h >
```

不会被预处理，而是被编译为（错误的）C代码。

如第2.1.2节所述，所有的源文件行（包括预处理器命令行）都可以在行末加个反斜杠（\）进行续行。这个操作发生在对预处理器命令进行扫描之前。

例子

下面的预处理器命令

```
#define err(flag,msg)  if (flag) \
   printf(msg)
```

与下面的命令相同

```
#define err(flag,msg)  if (flag) printf(msg)
```

如果下面第1行的行末添加了一个反斜杠字符，则这两行

```
#define BACKSLASH \
#define ASTERISK *
```

将被看成是单个预处理器命令

```
#define BACKSLASH #define ASTERISK *
```

如第2.2节所解释的那样，预处理器会把注释当成空白，注释内部的行分隔符并不会终止预处理器命令。

参考：注释 第2.2节；行的终止和延续 第2.1节；标记 第2.3节。

3.3 定义和替换

预处理器命令#define使一个名称（标识符）被定义为预处理器的一个宏。一种称为宏体的标记序列与这个名称相关联。当宏的名称在程序的源文本或者在其他预处理器命令的参数中被确认时，它就被当做是对这个宏的调用，宏名被宏体的一份拷贝所代替。如果这个宏被定义为接受参数，则宏名后面的实际参数就用于替换宏体中的形式参数。

例子

如果带有2个参数的宏sum被定义为

```
#define sum(x,y)  x+y
```

则预处理器就会对源程序行

```
result = sum(5,a*b);
```

执行简单的文本替换（也许并不是程序员所预想的）

```
result = 5+a*b;
```

由于预处理器并不会区分保留字和其他标识符，因此在原则上使用C的保留字作为预处理器的宏名也是可行的。但是，这种做法通常是一种不良的编程实践。预处理器不会认识注释、字符串或字符常量或#include文件名内部的宏名。

3.3.1 类似对象的宏定义

#define命令具有两种形式，取决于被定义的宏名后面是不是紧随一个左括号。最简单的类似对象的格式没有左括号：

```
#define  名称  标记序列opt
```

类似对象的宏不接受参数。它只是通过名称来调用的。在源程序文本中遇到宏名时，这个宏名就被宏体（即相关联的标记序列，可能为空）所替换。#define命令的语法并不要求在宏名后面使用等号或其他任何特殊的分隔符标记。宏体就是在宏名之后开始的。

类似对象的宏特别适用于在程序中引入名称常量，使诸如表的长度这样的“神奇数字”可以在一个地方编写，然后通过名称在其他地方被引用。这样，以后修改这个数字就非常方便了。

类似对象的宏的另一个重要用途是在外部所定义的函数和变量上隔离因编译器而异的限制。第2.5节提供了这方面的一个例子。

例子

下面是一些典型的宏定义：

```
#define BLOCK_SIZE 0x100
#define TRACK_SIZE (16*BLOCK_SIZE)
#define EOT '\004'
#define ERRMSG "*** Error %d: %s.\n"
```

一个常见的编程错误是错误地包含了一个等号：

```
#define NUMBER_OF_TAPE_DRIVES =  5  /* Probably wrong. */
```

这是合法的定义，但它导致NUBER_OF_TAPE_DRIVES被定义为“= 5”而不是“5”。如果接着编写了下面的代码片段：

```
if (count != NUMBER_OF_TAPE_DRIVES) …
```

它将被展开为：

```
if (count != = 5) …
```

它在语法上是非法的。基于相似的原因，程序员还要避免使用不必要的分号：

```
#define NUMBER_OF_TAPE_DRIVES 5 ;  /* Probably wrong. */
```

参考：复合赋值操作符 第7.9.2节；操作符和分隔符 第2.4节。

3.3.2 定义带参数的宏

更为复杂的、与函数相似的宏定义在一对括号内声明了用逗号分隔的形式参数的名称：

```
#define 名称( 标识符列表opt ) 标记序列opt
```

其中，标识符列表是个逗号分隔的形式参数名称列表。在C99中，标识符列表中还可以出现省略号（...，三个点），表示可变长度的参数列表。这个问题将在第3.3.10节讨论。在此之前，我们只考虑固定数量的参数列表。

左括号必须紧随宏名之后，中间不能有空格。如果宏名和左括号之间被一个空格所分隔，则这个宏被定义为不接受任何参数，并且宏体从这个左括号开始。

形式参数的名称必须是标识符，并且必须各不相同。宏体内并不一定要提到任何参数的名称（尽管一般都会提到）。类似函数的宏可以具有空的形参列表（即0个形式参数）。这种类型的宏适用于模拟不接受任何参数的函数。

类似函数的宏所接受的实参数量与形参相同。调用这种宏的形式是宏名、一个左括号、与每个形参对应的实际参数标记序列以及一个右括号。实际参数标记序列是由逗号分隔的（当一个不接受任何参数的类似函数的宏被调用时，必须提供一个空的实际参数列表）。当一个宏被调用时，宏名和左括号之间或者实际参数之间可以出现空格（有些旧式的或者有缺陷的预处理器实现并不允许实际参数标记列表延续多行，除非这些行的行末以用一个“\”符号进行续行）。

实际参数标记序列可以包含括号，只要它们被正确地嵌套并且左右括号的数量平衡。并且，它可以包含逗号，只要每个逗号出现在一组括号内部（这个限制是为了防止这些逗号与分隔实际参数的逗号混淆）。类似，花括号和方括号也可以出现在宏的参数中，但它们不能包含逗号，并且并不一定要平衡。出现在字符常量和字符串常量内部的括号和逗号不会在计算括号平衡时被考虑，也不会分隔实际参数。

在C99中，宏的参数可以是空的，也就是不包含任何标记。

例子

下面是一个把两个参数相乘的宏定义：

```
#define product(x,y)  ((x)*(y))
```

在下面这条语句中，它被调用了两次：

```
x = product(a+3,b) + product (c, d);
```

product宏的参数可以是函数调用（或宏调用）。函数参数列表中的逗号并不会影响宏参数的解析：

```
return product( f(a,b), g(a,b) ); /* OK */
```

例子

getchar宏具有空的形参列表：

```
#define getchar()  getc(stdin)
```

当它被调用时，就向它提供空的实参列表：

```
while ((c=getchar()) != EOF) …
```

（getchar、stdin和EOF是在标准头文件stdio.h中定义的。）

例子

我们还定义了一个接受一条任意的语句为参数的宏：

```
#define insert(stmt)  stmt
```

下面这个调用

```
insert( {a=1; b=1;} )
```

能够正确地工作。但是，如果我们把两条赋值语句修改为一条包含两个赋值表达式的语句：

```
insert( {a=1, b=1;} )
```

则预处理器则会显示错误信息，表示提供了太多需要插入的宏参数。为了修正这个问题，我们可以把它写为：

```
insert( {(a=1, b=1);} )
```

例子

定义在语句上下文环境中使用的类似函数的宏可能存在一些陷阱。下面这个函数交换它的两个参数x和y的值。假定这两个参数的类型可以转换为unsigned long并且在转换回去之后参数的值不会发生变化，并且不涉及标识符_temp。

```
#define swap(x, y) { unsigned long _temp=x; x=y; y=_temp; }
```

问题在于在swap后面加上一个分号是一种很自然的做法，就像真的把swap看成是个函数一样：

```
if (x > y) swap(x, y); /* Whoops! */
else x = y;
```

这将产生错误，因为宏展开包含了一个额外的分号（第8.1节）。下面，我们把展开后的语句放在单独一行中，更清晰地显示这个问题：

```
if (x > y) { unsigned long _temp=x; x=y; y=_temp; }
;
else x = y;
```

避免这个问题的一个较为聪明的方式是把宏体定义为一条do-while语句，后者可以接受在末尾添加分号（第8.6.2节）。

```
#define swap(x, y) \
   do { unsigned long _temp=x; x=y; y=_temp; } while (0)
```

当遇到一个类似函数的宏调用时，在参数处理之后，整个宏调用都被替换为经过处理的宏体的一份拷贝。参数处理是按下面的方式进行的。实际参数标记字符串与对应的形式参数名称相关联。此时将创建宏体的一份拷贝，其中每个形式参数的名称被与它相关联的实际参数标记序列所代替。然后，这份宏体拷贝就替换了这个宏调用，用经过处理的宏体的一份拷贝替换宏调用的过程称为*宏展开*。经过处理的宏体的拷贝称为该宏调用的*展开*。

例子

考虑这个宏定义，它提供了一种常规的方式，通过一个循环，从一个特定值到某个限值（含此限值）进行计数：

```
#define incr(v,low,high)  \
   for ((v) = (low); (v) <= (high); (v)++)
```

为了打印出1-20的整数的立方表，可以编写下面的代码：

```
#include <stdio.h>
int main(void)
{
   int j;
   incr(j, 1, 20)
      printf("%2d %6d\n", j, j*j*j);
   return 0;
}
```

对incr宏的调用被展开为下面这个循环：

```
    for ((j) = (1); (j) <= (20); (j)++)
```

多余的括号保证了复杂的实际参数并不会被编译器错误地解释（参见第3.3.6节）。

参考：do语句 第8.6.2节；语句的语法 第8.1节；unsigned long 第5.1.2节；空格 第2.1.2节。

3.3.3 宏表达式的重新扫描

当一个调宏用被展开之后，对宏调用的扫描就在宏展开的开始位置继续，使宏的名称在被展开宏的内部可以被认识，以便实现进一步的宏替换。宏替换并不是在#define命令的任何部分中执行的，甚至也不是在宏体中执行的，而是在处理命令以及定义宏名时执行的。对于某些特定的宏调用，只有当宏体被展开之后，宏体中的宏名才能被认识。

当一个类似函数的宏调用被扫描时，它的实际参数标记字符串也不会执行宏替换。只有当对展开的宏体重新进行扫描时，预处理器才能认识实际参数标记字符串中的宏名，前提是对应的形式参数在宏体中实际出现了1次或多次（因此导致实际参数标记字符串在展开的宏体中出现1次或多次）。

例子

假设有下面的定义：

```
#define plus(x,y)  add(y,x)
#define add(x,y)   ((x)+(y))
```

下面这个调用：

```
plus(plus(a,b),c)
```

是按照下面的方式展开的。

步骤	结果
1（最初）	plus (plus (a, b), c)
2	add (c, plus (a, b))
3	((c) + (plus (a, b)))
4	((c) + (add (b, a)))
5（最终）	((c) + (((b) + (a)))

在标准C中，出现在自身的展开体中的宏（直接出现或者通过某些嵌套宏展的中间序列）并不会被重新展开。这就允许程序员根据一个宏的旧定义对它进行重新定义。旧式的C预处理器在传统中并不会检测这种递归，因此试图继续展开，直到被某种系统错误所终止。

例子

下面这个宏修改平方根函数的定义，以一种不同常规的方式处理负的参数：

```
#define sqrt(x)  ((x)<0 ? sqrt(-(x)) : sqrt(x))
```

在标准C中，只要这个宏的求值次数不超过1次，它的效果就和预期的相同。但是，它在旧式的编译器中可能会产生错误。下面这个宏也有类似的问题。

```
#define char unsigned char
```

读者可以参见第7.4.3节中一个使用宏来追踪函数调用的有趣例子。

3.3.4 预定义的宏

标准C的预处理器需要定义一些类似对象的宏（表3-2）。这些宏的名称都是以两个下划线字符开始和结束的。程序员不能取消这些预定义宏的定义（使用#undef）或对它们进行重新定义。

__LINE__和__FILE__宏适用于打印某些类型的错误信息。__DATE__和__TIME__宏可以记录编译的时间。__DATE__和__TIME__的宏在编译时为常量值。__LINE__和__FILE__宏的值是由编译器确定的，但受#line指令（第3.6节）更改的控制。C99的预定义标识符__func__（第2.6.1节）的目的与__LINE__相似，但它实际上是个代码块作用域变量，而不是一个宏。它提供了外层函数的名称。

__STDC__和__STDC__VERSION宏适用于编写与标准C和非标准C编译器兼容的代码。__STDC__HOSTED__宏是C99所引入的，用于区分宿主环境和独立环境。其余的C99宏提示了编译器遵循其他相关的国际标准的浮点数和宽字符工具（推荐遵循这些标准，但并不是必须）。

表3-2 预定义的宏

宏	值
__LINE__	当前源程序行的行号，用十进制整数常量表示
__FILE__	当前源文件的名称，用字符串常量表示
__DATE__	编译时的日历日期，用“Mmm dd yyyy”形式的字符串常量表示。Mmm是由asctime产生的
__TIME__	编译时的日历时间，用“hh:mm:ss”形式的字符串常量表示，后者是由asctime返回的
__STDC__	当且只当编译器遵循ISO标准时，它的值是十进制常量1
__STDC__VERSION__	如果编译器遵循C89修正案1，则这个宏的值是199409L。如果编译器遵循C99，则这个宏的值是199901L。在其他情况下，这个宏的值是未定义的

（续）

宏	值
__STDC__HOSTED__	（C99）如果当前是宿主系统，这个宏的值是1。如果当前是独立系统，这个宏的值是0
__STDC__IEC__559__	（C99）如果浮点实现遵循IEC 60599标准，这个宏的值为1，否则为未定义
__STDC__IEC__559__COMPLEX__	（C99）如果复数运算的实现遵循IEC 60559，则这个宏的值为1，否则这个宏的值是未定义的
__STDC__ISO10646__	（C99）定义为一个长整数常量，yyyymmL表示wchar_t值遵循进行了修正和增补的ISO 10646标准，就像特定的年和月一样。否则为未定义

a 这些宏在非ISO的编译器中很常用。

编译器一般会定义其他的宏，与环境的相关信息进行通信，例如程序被编译时所在的计算机的类型。至于哪些宏被定义则取决于编译器，尽管UNIX系统一般预定义了unix。和内置的宏不同，这些宏可以被取消定义。标准C要求编译器特定的宏名以一个前导的下划线开始，后面可以是一个大写字母或另一个下或线（unix宏并不符合这个标准）。

例子

预定义宏适用于一些类型的错误信息：

```
if (n != m)
    fprintf(stderr,"Internal error: line %d, file %s\n",
        __LINE__, __FILE__ );
```

还有一些编译器所定义的宏可以用于隔离宿主或目标特定的代码。例如，Microsoft Visual C++把__WIN32定义为1：

```
#ifdef __WIN32
    /* Code for Win32 environment */
#endif
```

__STDC和__STDC__VERSION__宏适用于编写那些需要兼容标准C和非标准C编译器的程序：

```
#ifdef __STDC__
  /* Some version of Standard C */
#if defined(__STDC_VERSION__) && __STDC_VERSION__>=199901L
    /* C99 */
#elif defined(__STDC_VERSION__) && __STDC_VERSION__>=199409L
    /* C89 and Amendment 1 */
#else
    /* C89 but not Amendment 1 */
#endif
#else /* __STDC__ not defined */
  /* Not Standard C */
#endif
```

参考：asctime工具 第20.3节；复数运算 第23章；fprintf 第15.11节；独立系统和宿主系统 第1.4节；#ifdef 预处理器命令 第3.5.3节；#if预处理器命令 第3.5.1节；取消宏的定义 第3.3.5节；wchar_t 第24.1节。

3.3.5 取消宏定义和重新定义宏

#undef命令可以取消定义一个名称为宏：

```
#undef name
```

这个命令使预处理器忘记name的所有宏定义。取消一个当前未定义宏的定义并不是错误。当一个名称被取消定义之后，就可以向它提供一个全新的定义（使用#define），而不会产生任何错误。在#undef命令内部，并不会执行宏替换。

标准C以及其他许多编译器允许对宏进行温和的重定义。也就是说，新定义与原来的定义必须逐标记相同。新定义所包含的空白字符的位置必须与原定义相同，尽管具体的空白字符可以不同。我们认为程序员不应该依赖这种温和的重定义。更好的风格是让所有的程序入口都使用同一个单独的定义点，包括宏（有些旧式的C编译器可能不允许任何类型的重定义）。

例子

在下面的定义中，NULL的重定义是允许的，但对FUNC的所有重定义都是非法的（第1个重定义包含了原定义中未曾出现的空格，第2个重定义修改了两个标记）。

```
# define NULL 0
# define FUNC(x) x+4
# define NULL /* null pointer */ 0
# define FUNC(x) x + 4
# define FUNC(y) y+4
```

例子

当程序员由于法律的原因无法知道是否存在以前的定义时，可以使用#ifndef命令测试是否存在一个现有的定义，以避免对它进行重定义：

```
#ifndef MAXTABLESIZE
#define MAXTABLESIZE 1000
#endif
```

在那些允许命令中的宏定义调用C编译器的实现中，这种用法特别有用。例如，下面这个UNIX对C的调用提供了5000作为MAXTABLESIZE宏的初始定义。随后，C程序员应该像前面一样检查这个定义：

```
cc -c -DMAXTABLESIZE=5000 prog.c
```

尽管标准C并不允许，但一些旧式的预处理器实现在处理#define和#undef时就像维护一个定义堆栈一样。当一个名称用#define进行重新定义时，它的旧定义被压入到堆栈中，然后用新定义替换这个旧定义。当一个名称用#undef取消定义时，当前的定义就被丢弃，最近的一个定义（如果有）就被恢复。

参考：#define命令 第3.3节；#ifdef和ifndef命令 第3.5.3节。

3.3.6 宏展开的优先级错误

宏的操作完全是标记的文本替换。只有在宏展开过程结束之后，才会把宏体解析为声明、表达式或语句。如果不加注意，就有可能导致令人吃惊的结果。作为规则，对出现在宏体内部的每个参数加上括号总是最安全的。如果整个宏体在语法上是一个表达式，它的两边也应该加上括号。

例子

考虑下面这个宏定义：

```
#define SQUARE(x) x*x
```

它的思路是SQUARE接受一个参数表达式，并产生一个新的表达式，计算这个参数的平方。例如，SQUARE(5)被展开为5*5。但是，表达式SQUARE(z + 1)被展开为z + 1 * z + 1，解析的结果是z + (1 * z) + 1，而不是(z + 1) * (z + 1)。下面这个SQUARE定义可以避免这个问题：

```
#define SQUARE(x) ((x)*(x))
```

为了防止对诸如(short) SQUARE(z + 1)这样的表达式作出错误解释，外层的括号是必要的。

参考：类型转换表达式 第7.5.1节；表达式的优先级 第7.2.1节。

3.3.7 宏参数的副作用

宏也可能因为副作用而产生问题。由于宏的实际参数可能会被文本复制，它们的执行次数可能不止一次，因此实际参数的副作用可能会产生不止一次。反之，与宏调用看上去很相似的真正函数调用对实参表达式的求值次数正好只有一次，因此表达式的副作用也正好出现一次。为了避免这种问题，在使用宏时必须小心。

例子

考虑前一个例子的SQUARE宏以及（几乎）完成相同任务的square函数：

```
int square(int x) { return x*x; }
```

这个宏可以求浮点数的平方根，而这个函数只能求整数的平方根。另外，调用函数的运行速度要稍慢于调用宏。但是，和可能产生的副作用相比，这些差别是微不足道的。在下面的程序片段中：

```
a = 3;
b = square(a++);
```

变量b获得的值是9，而变量a的最终值是4。但是，在下面这个表面看上去相似的程序片段中：

```
a = 3;
b = SQUARE(a++);
```

变量b值可能是12，变量a的最终值可能是5，这是因为后面这个程序片段被展开为：

```
a = 3;
b = ((a++)*(a++));
```

（12和5可能是b和a的最终值，这是因为标准C编译器可能以不同的方式对表达式((a++)*(a++)进行求值。参见第7.12节。）

参考：增值操作符++ 第7.4.4节。

3.3.8 把标记转换为字符串

标准C具有一种机制，把宏的参数（在展开之后）转换为字符串常量。在此之前，在许多C预处理器中，程序员必须依赖语言的一个空子，以不同的方式实现与此相同的效果。

在标准C中，出现在宏定义内部的#标记被当做单目的“字符串化”操作符，它的后面必须

是个宏形式参数的名称。在宏展开期间，#和形式参数的名称被一个对应的包含在双引号内的实际参数所代替。在创建字符串时，实参的标记列表中的每个空白字符序列被单个空格字符所代替，所有内嵌的引号或反斜杠字符前面都加上一个反斜杠，以保留它们在字符串中的原来含义。参数起始和结束位置的空白字符被忽略，因此空的实参（即使是逗号分隔的空白字符）被展开为空字符串""。

例子

考虑TEST宏的标准C定义：

```
#define TEST(a,b) printf( #a "<" #b "=%d\n", (a)<(b) )
```

语句TEST(0, 0xFFFF)；TEST('\n', 10)；将被展开为

```
printf("0" "<" "0xFFFF" "=%d\n", (0)<(0xFFFF) );
printf("'\\n'" "<" "10" "=%d\n", ('\n')<(10) );
```

在相邻的字符串连接之后，它们成为：

```
printf("0<0xFFFF=%d\n", (0)<(0xFFFF) );
printf("'\\n'<10=%d\n", ('\n')<(10) );
```

一些非标准C编译器将会替换字符串和字符常量“内部”的宏形式参数。标准C禁止这种做法。

例子

在不遵循标准C的编译器中，可以按照下面的方式编写TEST宏：

```
#define TEST(a,b) printf( "a<b=%d\n", (a)<(b) )
```

展开TEST(0, 0xFFFF)的结果类似于字符串化的结果：

```
printf("0<0xFFFF=%d\n", (0)<(0xFFFF) );
```

展开TEST(0, 0xFFFF)时几乎肯定会丢掉那个额外的反斜杠，printf函数的输出将被预料之外的行分隔符所破坏：

```
printf("'\n'<10=%d\n", ('\n')<(10) );
```

在非ISO编译器中，空白字符的处理也可能因编译器而异，这也是避免依赖这个特征的原因之一，除非是在标准C编译器中。

3.3.9 宏展开中的标记合并

在标准C中，合并标记以形成新标记的行为是由宏定义中的合并操作符##所控制的。在宏替换列表中（在扫描更多的宏之前），##操作符两边的两个标记被合并为单个标记。##操作符的两边必须存在标记，它不能出现在替换列表的开始或结束位置。如果这些组合无法形成一个合法的标记，则结果是未定义的。

```
#define TEMP(i)  temp ## i
TEMP(1) = TEMP(2 + k) + x;
```

在预处理之后，它成为：

```
temp1 = temp2 + k + x;
```

在前面这个例子中，在展开TEMP() + x时，会出现一种奇怪的情况。这个宏定义是合法的，

但最终##的右边却没有标记（除非是+，但这并不是我们所需要的）。这个问题的解决方案是以特殊的方式处理形式参数i，使它根据##展开为一个特殊的“空”标记。因此，TEMP() + x的展开结果将是预期的temp + x。

标记连接不能用于产生统一字符名称。

和把宏参数转换为字符串（第3.3.8节）一样，在许多非标准C编译器中，程序员可以依赖C语言的一个空子来实现类似的合并功能。尽管C语言的最初定义明确地把宏体描述为标记序列而不是字符序列，然而许多C编译器在展开和重新扫描宏体时是把它们看成是字符序列。这样至少存在一个很显而易见的空子，就是在编译器处理注释并将它们全部删除时（而不是用一个空格替换它们）。程序员在编写程序时可以聪明地利用这个空子实现标记粘贴的效果。

例子

考虑下面这个例子：

```
#define INC ++
#define TAB internal_table
#define INCTAB table_of_increments
#define CONC(x,y) x/**/y
CONC(INC,TAB)
```

标准C把CONC的宏体解释为两个标记：x和y，并用一个空格分隔（注释被转换为空格）。CONC(INC, TAB)这个调用被展开为两个标记INC TAB。但是，有些非标准编译器简单地删除注释，然后重新在宏体中扫描标记，这样CONC(INC, TAB)就被展开为单个标记INCTAB。

步骤	标准C所执行的展开	非标准C可能执行的展开
1	CONC(INC,TAB)	CONC(INC,TAB)
2	INC/**/TAB	INC/**/TAB
3	INC TAB	INCTAB
4	++ internal_table	table_of_increments

参考：递值操作符++ 第7.5.8节；统一字符名称 第2.9节。

3.3.10 宏的可变参数列表

在C99中，类似函数的宏可以用一个省略号来表示它的最后一个形式参数或唯一的那个形式参数。它表示这个宏可以接受可变数量的实参。

```
#define 字(标记、符列表, ...) 标记序列
#define 名字 (...) 标记序列
```

当这样的一个宏被调用时，至少必须提供与identifier-list中的标识符一样多的实际参数。缀尾的参数（包括用于分隔的逗号）被合并为一个单独的预处理标记序列，称为可变参数。出现在宏定义的替换列表中的标识符__VA_ARGS__被看成是个宏形参，与它对应的实参就是合并后的可变参数。也就是说，__VA_ARGS__被额外参数列表（包括逗号分隔符）所替换，只有形参列表中存在...的宏定义中才能出现__VA_ARGS__。

具有可变数量的参数的宏经常用于作为接受可变数量的参数的函数的接口，例如printf。通过使用#字符串化操作符，它们还可以用于把一个参数列表转换为单个字符串，而不必把参数放在括号中。

例子

下面这些指令创建了一个my_printf宏，它可以把参数写入到标准错误流或标准输出流。

```
#ifdef DEBUG
#define my_printf(...) fprintf(stderr, __VA_ARGS__)
#else
#define my_printf(...) printf(__VA_ARGS__)
#endif
```

它可以按照下面这种方式使用：

```
my_printf("x = %d\n", x);
```

例子

假设有下面这个定义：

```
#define make_em_a_string(...) #__VA_ARGS__
```

下面这个调用

```
make_em_a_string(a, b, c, d)
```

被展开为字符串

```
"a, b, c, d"
```

3.3.11 其他问题

有些非标准编译器并不会对宏定义和宏调用执行严格的错误检查，包括允许宏体中的不完整标记由宏调用之后所出现的文本补充完整。有些编译器所缺乏的错误检查机制并不能被合理地利用。标准C重新确定了宏体必须是形式完整的标记序列。

例子

例如，在一个非ISO编译器中，下面这个片段

```
#define FIRSTPART "This is a split
…
printf(FIRSTPART string."); /* Yuk! */
```

在预处理之后，将会在源文本中产生

```
printf("This is a split string.");
```

3.4 文件包含

#include预处理器命令对指定源文本文件的完整内容进行预处理，用这些内容替换这条#include指令。在标准C中，#include命令具有下面这3种形式：

```
#include <h字符序列>
#include " q字符序列 "
#include 预处理器标记                    (标准C)
```

h字符序列：

除了 > 和行末字符之外的所有字符序列

q字符序列：

除了 "和行末字符之外的所有字符序列

预处理器标记：

任何C标记序列，或者无法被解释为标记的不是以<或"开头非空白字符

在前两种形式的#include指令中，定界符之间的字符应该是编译器所定义的某种形式的文件名。在结尾的>或"之后只能出现空白字符。所有的C编译器都支持这两种形式的#include指令。文件名受到标准C的三字符组替换和源代码行延续的影响，但除此之外不会再对文件名中的字符进行其他处理。

在第3种形式的#include指令中，预处理器标记采用常规的宏展开，展开结果必须与前两种形式之一匹配（包括引号或尖括号）。这种形式的#include较为少见，在非标准的编译器中可能并未实现，或者按照不同的方式所实现。

例子

下面是使用第3种形式的#include指令的一个例子：

```
#if some_thing==this_thing
#   define IncludeFile "thisname.h"
#else
#   define Includefile <thatname.h>
#endif
…
#include Includefile
```

这种风格可以用于本地化的惯用形式，但注重与旧式编译器保持兼容的程序员应该用#include指令代替前面所出现的#define指令：

```
#if some_thing==this_thing
#   include "thisname.h"
#else
#   include <thatname.h>
#endif
```

众所周知，文件名的语法依赖于操作系统，但是，标准C要求所有编译器允许出现在#include指令中的文件名由字母和数字组成（以字母开头），然后是个点号和单个字母。C99允许在点号之前至少出现8个字符和数字，但C89只保证点号前至少可以出现5个字母。这种“允许”意味着这种形式的文件名必须映射到一个依赖于系统的文件。

由双引号所定界的文件名和由尖括号所定界的文件名的区别在于编译器查找文件的方式。这两种形式都在一组由编译器所定义的位置（两者可能不同）搜索文件。一般情况下，下面的形式

```
#include <文件名>
```

根据编译器所定义的规则，在一些标准位置搜索被包含的文件。这些标准位置通常包含了编译器自己所提供的头文件，例如stdio.h。下面这种形式

```
#include "文件名"
```

也在一些标准位置搜索被包含的文件，但通常是在搜索了程序员所设置的当前目录之后才在这些位置进行搜索。在C语言之外，编译器常常提供了一些标准的方式指定搜索这些文件的位置。一般的思路是"..."形式表示程序员自己编写的头文件，而<...>形式表示编译器所提供的标准头文件。

事实上，像stdio.h这样的标准头文件在标准C中也是作为特殊情况处理的。标准C要求编译器认识#include命令中由< >定界的标准库头文件名，而不管这些文件名是否指定了真正的文件名。它们可以作为特殊情况处理，C编译器简单地“知道”它们的内容。出于这个原因，标准调用它们多个标准头文件，而不是一个标准头文件。在本书中，我们用到了两种方式。

被包含的文件可以包含#include命令。这种#include命令的嵌套深度是由编译器所指定的，但标准C要求至少支持8层（C99要求至少支持15层）嵌套。包含文件的位置可能会影响被嵌套文件的搜索规则。

例子

假设我们编译一个C程序first.c，它位于系统目录/near。first.c文件包含了下面这几行：

```
// In /near/first.c
#include "/far/second.h"
```

它指定了在/far目录寻找second.h文件。second.h文件包含了下面这几行：

```
// In /far/second.h
#include "third.h"
```

它并没有指定目录。编译器将选择最初的工作目录中的/near/third.h文件，还是选择包含它的目录中的/far/third.h文件呢？有些UNIX C编译器将找到/far/third.h。C的最初描述似乎建议找到的应该是/near/third.h文件。大多数编译器让程序员指定进行搜索的目录列表，用于那些未指定目录的包含文件。

参考：字符串常量 第2.7.4节；三字符组 第2.1.4节。

3.5 条件编译

预处理器的条件编译指令允许预处理器根据一个经过计算所得出的条件传递或删除几行源文本代码。

3.5.1 #if、#else和#endif命令

下面这些预处理器命令一起使用时，允许在编译时根据#if、#else和#endif所设置的条件包含或排除部分源文本行。它们是按照下面的方式使用的：

```
#if   常量表达式
        文本行组1
#else   文本行组2
#endif
```

常量表达式受宏替换的影响，它的求值结果必须是个算术常量值。第7.1.1节讨论了对表达式的一些限制。“文本行组”可以是任何数量、任何类型的文本行，甚至是其他预处理器文本行或者根本没有任何文本行。#else命令以及它后面的文本行组可以被省略，相当于#else命令后面没有任何文本行。在文本行组中还可以包含一组或多组#if - #else - #endif命令。

像前面这样的一组命令进行处理的结果是一组文本行被传递给编译器进一步处理，而另一组文本行则被丢弃。首先，#if命令中的常量表达式被求值。如果它的值不是0，则“文本行组1”被传递给编译器进行编译，而“文本行组2”（如果存在）则被丢弃。否则，“文本行组1”就被丢弃。并且，如果存在#else命令，则“文本行组2”被传递给编译器进行编译。但是，如果不存

在#else命令，就没有任何文本组行被传递给编译器。第3.5.4节和第7.11节详细描述了可以在#if命令中使用的常量表达式。

被丢弃的文本行组不会被预处理器处理。它内部的宏替换不会被执行，预处理器命令也被忽略。唯一的例外是在被丢弃的文本行组内部，将会确认#if、#else、#ifndef、#elif、#else和#endif命令，以便对它们进行计数（这是唯一的目的）。维护条件编译命令的正确嵌套是有必要的。这意味着被丢弃的行也会被扫描并分解为标记，并且能够确认其中的字符串常量和注释，以便对它们进行正确的定界。

如果#if或#elif的常量表达式中出现了一个未定义的宏名，它就被整数常量0所替换。这意味着“#ifdef 名称”这条命令和“#if 名称”这条命令将具有相同的效果，只要被定义的宏名具有非零的算术常量值。我们认为在这种情况下使用#ifdef或预定义的操作符是更好的方法，但标准C也支持#if的这种用法。

参考：defined 第3.5.5节；#elif 第3.5.2节；#ifdef 第3.5.3节。

3.5.2 #elif命令

#elif命令在标准C编译器以及那些较为现代的非ISO编译器中存在。它是一种很方便的命令，因为它能够简化一些预处理器条件。这是按照下面的方式使用的：

```
#if   常量表达式1                    (或#ifdef，或#ifndef)
        文本行组1
#elif  常量表达式2
        文本行组2
...
#elif 常量表达式n
        文本行组n
#else
        最后一组文本行
#endif
```

这些命令序列的处理方式是最多只有一组文本行被传递给编译器进行编译，其他各组文本行都被丢弃。首先，#if命令中的常量表达式1被求值。如果它的值不是0，则“文本行组1”被传递给编译器进行编译，所有其他到匹配的#endif为止的各组文本行都被丢弃。如果#if命令中的常量表达式1的值不是0，则第1条#elif语句中的常量表达式2被求值。如果这个值不是0，则“文本行组2”被传递给编译器进行编译。在一般情况下，每个常量表达式i按顺序被求值，直到有一个产生了非零值。然后，预处理器就把产生这个非零值的常量表达式所在的命令后面的文本行组传递给编译器，并忽略其他命令的常量表达式，并丢弃其他所有文本行组。如果没有常量表达式i为非零值，并且存在一条#else语句，则#else命令后面的那组文本行被传递给编译器。但是，如果此时不存在#else命令，则不会有任何文本组行被传递给编译器。在#elif命令中所使用的常量表达式可以与#if命令中所使用的常量表达式相同（参见第3.5.4节和第7.11节）。

在一组被丢弃的文本行中，#elif命令就像#if、#else、#ifndef、#else和#endif一样被确认，唯一的目的是为了对它们进行计数。维护条件编译命令的正确嵌套是十分必要的。

宏替换是在#elif命令后面的一条命令中执行的，因此可以在常量表达式中使用宏调用。

例子

尽管在合适的时候使用#elif命令是非常方便的，但也可以只使用#if、#else和#endif来实现它

的功能。下面显示了一个例子。

使用#elif	不使用#elif
#if 常量表达式1 文本行组1 #elif 常量表达式2 文本行组2 #else 最后一组文本行 #endif	#if 常量表达式1 文本行组1 #else #if 常量表达式2 文本行组2 #else 最后一组文本行 #endif #endif

3.5.3 #ifdef和#ifndef命令

#ifdef和#ifndef命令可以用于测试一个名称是否被定义为一个预处理器宏。下面这种形式的命令行

```
#ifdef  name
```

如果name已经被定义为一个预处理器宏，它的含义相当于

```
#if  1
```

如果name没有被定义为预处理宏或者被#undef命令取消了定义，则它相当于

```
#if  0
```

#ifndef命令的含义正好相反。当name未被定义时，它的测试结果为真，否则为假。

注意，#ifdef和#ifndef只测试名称是否由#define命令所定义（或者由#undef取消定义），它们与被编译的C程序文本的声明中所出现的名称无关（有些C编译器允许使用特殊的编译器命令行参数来定义名称）。

例子

#ifndef和#ifdef可以按照几种不同的风格在C程序中使用。首先，通过一组只有一个被定义的符号实现一种预处理器时的枚举类型是常见的做法。例如，假设我们希望使用VAX、PDP11和GRAY2这组名称来指示程序被编译时所在的计算机。有些人坚持认为所有这些名称都应该被定义，其中只有1个被定义为1，其余的都定义为0：

```
#define VAX 0
#define PDP11 0
#define CRAY2 1
```

然后，可以按照下面的方式选择需要编译的依赖于计算机的源代码：

```
#if VAX
    VAX-dependent code
#endif
#if PDP11
    PDP11-dependent code
#endif
#if CRAY2
    CRAY2-dependent code
#endif
```

但是，习惯的做法是只定义一个符号：

```
#define CRAY2 1
    /* None of the other symbols is defined. */
```

然后用条件编译指令测试这些符号是否被定义：

```
#ifdef VAX
    VAX-dependent code
#endif
#ifdef PDP11
    PDP11-dependent code
#endif
#ifdef CRAY2
    CRAY2-dependent code
#endif
```

例子

#ifdef和#ifndef命令的另一个用途是提供宏的默认定义。例如，一个库文件可能只有在一个名称并没有被定义的情况下才会向它提供定义：

```
#ifndef TABLE_SIZE
#define TABLE_SIZE 100
#endif
...
static int internal_table[TABLE_SIZE];
```

程序可以简单地包含这个文件：

```
#include <table.h>
```

在这种情况下，TABLE_SIZE的定义将会是100，不管是在库文件中还是在#include之后。或者，程序可以提供一个显式的定义：

```
#define TABLE_SIZE 500
#include <table.h>
```

在这种情况下，TABLE_SIZE的定义将始终是500。

用“#if name”代替“#ifdef name”或“#ifndef name”来测试一个名称是否被定义是一种常见的C的编程错误。这种不正确的形式经常能够实现目的，因为预处理器用常量0来替换#if表达式中并没有被定义为宏的任何名称。因此，如果name没有被定义，这3种形式都是一样的。但是，如果name被定义为0这个值，即使这个名称已经被定义，“#if name”的结果仍然是假。类似，如果用一个非法的表达式来定义name，则“#if name”将导致错误。

参考：#define 第3.3节；defined操作符 第3.5.5节；#include 第3.4节；预处理器词法约定 第3.2节；#undef 第3.3节。

3.5.4 条件命令中的常量表达式

第7.11.1节描述了可以在#if和#elif命令中使用的表达式。它们包括整数常量以及所有的整数算术、关系、位和逻辑操作符。

C99强迫所有的预处理器根据目标计算机上的最大整数类型来执行算术运算，它是stdint.h所定义的intmax_t或uintmax_t。以前，标准C并不要求编译器实现目标计算机的运算属性。

参考：intmax_t 第21.5节；uintmax_t 第21.5节。

3.5.5 defined操作符

defined操作符可以在#if和#elif表达式中使用，而不能用于别处。下面这两种形式的表达式：

```
defined name
defined( name )
```

如果name在预处理器中已经被定义，上面的表达式的值为1，否则为0。

例子

defined命令允许程序员用下面的指令

```
#if defined(VAX)
```

代替

```
#ifdef VAX
```

defined操作符用起来可能更为方便，因为它可以创建像下面这样复杂的表达式：

```
#if defined(VAX) && !defined(UNIX) && debugging
…
```

3.6 显式的行号

预处理器命令#line向C编译器提示了源程序是由其他工具所生成的，并提示源程序中的位置与用户最初所编写的文件（C源程序是通过这个文件形成的）中的行的对应关系。#line命令可以使用两种形式之一。下面这种形式

```
# line n " filename "
```

表示源代码的下一行来自用户所编写的称为filename的文件的第n行。n必须是个十进制数字序列。下面这种形式

```
# line n
```

表示源代码的下一行来自#line命令上次所提到的用户所编写的文件的第n行。最后，如果#line命令并不与前面的任一形式匹配，则它被解释为

```
# line  预处理器标记
```

在参数标记序列中将会执行宏替换，其结果必须与前面两种#line格式之一匹配。

#line命令所提供的信息用于设置预定义宏__LINE__和__FILE__的值。如果用于其他情况，则它的行为是未指定的，并且编译器可以忽略它。一般情况下，这种信息也可以出现在诊断信息中。有些生成C源代码文本作为输出的工具将使用#line，使错误信息可以与这个工具的输入文件而不是实际的C源文件相关。

有些C编译器允许预处理器独立于编译器的其他部分使用。事实上，有时候预处理器是个独立的程序，它执行之后产生一个中间文件，供真正的编译器继续进行处理。在这种情况下，预处理器可能会在中间文件中产生新的#line命令。然后，编译器可以认识这些行号，即使它们可能无法被其他的预处理器命令所认识。预处理器是否生成#line命令取决于C的实现。类似，预处理器是否在输入中传递、修改或消除#line命令也是取决于C的实现。

旧版本的C允许简单地使用“#”作为#line命令的同义词，允许采用下面这种形式：

```
# n filename
```

这种语法被认为是过时的，标准C不再允许这种用法。但是，由于兼容性的缘故，许多编译器继续支持这种做法。

参考：__FILE__ 第3.3.4节；__LINE__ 第3.3.4节。

3.7 pragma指令

#pragma命令是标准C新增的。下面的命令名后面可以出现任何标记序列：

```
# pragma 预处理器标记
```

#pragma指令可以由C的实现所使用，以添加新的预处理器功能或者向编译器提供因实现而异的信息。#pragma命令后面所出现的信息并无任何限制，编译器（包括预处理器）应该忽略它们所不理解的信息。除了在C99标准pragma（3.7.1节）的情况下，#pragma后的标记是否受宏扩展的影响取决于实现。

显然，不同的编译器可能会以不同的方式理解相同的信息。因此，以条件编译的形式，根据具体使用的编译器来选择需要使用的#pragma指令是明智的做法。

例子

下面的代码检查在发布#pragma命令之前是否使用了正确的编译器（tcc）、计算机以及遵循标准的C实现：

```
#if defined(_TCC) && defined(__STDC__) && defined(vax)
#pragma builtin(abs),inline(myfunc)
#endif
```

参考：defined 第3.5.3节；内存模型 第6.1.5节；#if 第3.5.1节。

3.7.1 标准pragma命令

C99引入了一些pragma指令，表示一些特殊的含义。为了区分它们，所有的标准pragma指令都必须加上前缀STDC（在任何具体实现宏扩展之前）。也就是说，下面这条指令

```
#pragma FENV_ACCESS ON
```

是一条由编译器所定义的pragma指令，但下面的指令

```
#pragma STDC FENV_ACCESS ON
```

指定了C99的pragma指令FENV_ACCESS。如果一个标准的pragma名称前面没有前缀STDC，编译器将会发出一条友好的警告信息，因为这很可能是个常见的错误。

C99所定义的仅有的标准pragma指令是FP_CONTRACT、FENV_ACCESS和CX_LIMITED_RANGE。它们都接受一个“打开-开闭”的开关作为参数。

打开-关闭开关：

```
ON
OFF
DEFAULT
```

DEFAULT参数把pragma指令设置为它的初始默认值（打开或关闭）。每个标准pragma都指

定了一个默认值（有时候，它的默认值是由编译器所定义的）。

参考：CX_LIMITED_RANGE 第23.2节；FENV_ACCESS 第22.2节。

3.7.2 标准pragma指令的位置

标准pragma指令必须遵循一定的位置规则，这样pragma指令更易于处理，并且允许pragma的嵌套。标准pragma指令可以出现在两个位置：在翻译单元的顶层，在任何外部声明之前；或者在复合语句的起始位置，在所有的显式声明和语句之前。

当位于顶层时，pragma指令的效果一直维持到当前翻译单元的末尾或者直到遇见了同一条pragma指令的另一个实例。后者可能是顶层的另一条pragma指令（此时它会覆盖第1条pragma指令），也可能是一条复合语句中的pragma指令。

当pragma指令位于复合语句的开始位置时，它将一直有效，直到该复合语句的结束（从词法上），或者直到在该复合语句中遇到同一条pragma指令的另一个实例。后者可能位于同一条复合语句的起始位置（此时它会覆盖第1条pragma指令），或者位于另一条内层复合语句中。在包含一条标准pragma指令的复合语句结束时，这个pragma将会恢复到该复合语句之前它的状态。也就是说，嵌套的标准pragma将遵循常规的变量作用域规则，区别在于它们可以在同一个作用域层次上被多次指定。

参考：作用域 第4.2.1节。

3.7.3 _Pragma操作符

C99增加了_Pragma操作符，以便更灵活地利用pragma工具。在宏展开之后，下面这种形式的操作符表达式

```
_Pragma ("字符串常量")
```

的处理方式是把字符串常量的内容（在删除两边的双引号，并把字符串常量内部的\"替换为"，把\\替换为\之后）看成是#pragma指令中所出现的预处理器标记。例如，下面这个表达式

```
_Pragma("STDC FENV_ACCESS ON")
```

将被看成是在这个位置上出现了下面这条pragma指令。

```
#pragma STDC FENV_ACCESS ON
```

#pragma指令本身必须出现在同一行中，而且它的预处理器标记并不会执行宏展开，但_Pragma可以被其他表达式所包围，并且可以通过宏展开而产生。

3.8 错误指令

#error指令是标准C新增的。它的后面可以出现任何标记序列：

```
#error  预处理器标记
```

#error指令产生一条编译时错误信息，其中包含了参数中的标记，而后者又受宏展开的影响。

例子

#error指令特别适用于检测程序员对预处理期间的限制的不一致行为和违规行为：

```
#if defined(A_THING) && defined(NOT_A_THING)
#error Inconsistent things!
#endif
```

```
#include "sizes.h"    /* defines SIZE */
...
#if (SIZE % 256) != 0
#error "SIZE must be a multiple of 256!"
#endif
```

在第1个#error例子中，我们并没有使用字符串常量。在第2个例子中，我们使用了字符串常量，这是因为我们并不想让SIZE标记在输出信息中展开。

参考：defined 第3.5.3节；#if 第3.5.1节。

3.9 C++兼容性

C++使用C89预处理器，因此C和C++在这方面的差别极微。

预定义宏

C++编译器预定义了__cplusplus宏，可以在源文件中使用，并且在C和C++环境中都适用。这个预定义宏并不遵循标准C的拼写约定，而是与原有的C++编译器兼容。在标准C++中，它的值是个版本号，例如199711L。

至于C++环境（在C++的当前定义中）是否定义了__STDC__，则是由编译器定义的。标准C和标准C++之间存在足够的差别，因此并不清楚是否应该定义__STDC__。

表3-2所列出的只有C99才有的宏在C++中均不存在。

例子

为了与传统C、标准C和C++兼容，应该按照下面这种方式对环境进行测试：

```
#ifdef __cplusplus
    /* It's a C++ compilation */
#else
#ifdef __STDC__
    /* It's a Standard C compilation */
#else
    /* It's a non-Standard C compilation */
#endif
#endif
```

如果读者知道自己所使用的C编译器遵循标准C，则可以采用简化的方式：

```
#if defined(__cplusplus)
    /* It's a C++ compilation */
#else
    /* It's a Standard C compilation */
#endif
```

参考：__STDC__ 第3.3.4节；__STDC__VERSION__ 第3.3.4节。

3.10 练 习

1. 下面哪个标准C的宏定义（很可能）是错误的？为什么？哪些定义在传统C中可能导致问题？

(a) `#define ident (x) x`
(b) `# define FIVE = 5;`
(c) `#define PLUS +`
(d) `#define void int`

2. 下面是一些宏定义和宏调用。每个宏调用分别会被标准C和传统C怎样展开呢？

	定义	调用
(a)	`#define sum(a,b) a+b`	`sum(b,a)`
(b)	`#define paste(x,y) x/**/y`	`paste(x,4)`
(c)	`#define str(x) # x`	`str(a book)`
(d)	`#define free(x )x ? free(x) : NULL`	`free(p)`

3. 下面显示了两个头文件和一个C程序文件。如果C预处理器对这个程序文件进行处理，会产生什么结果？

```
/* File blue.h */
int blue = 0;
#include "red.h"
```

```
/* File red.h */
#ifndef __red__
#define __red__
#include "blue.h"
int red = 0;
#endif
```

```
/* File test.c */
#include "blue.h"
#include "red.h"
```

4. 有人展示了下面这个宏定义，它的用意是对它的数值参数进行加倍。这个宏有什么错误？改写这个宏，使它能够执行正确的操作。

```
#define DBL(a)    a+a
```

5. 在下面这个标准C程序片段中，M(M)(A,B)展开后的结果是什么？

```
#define M(x)      M ## x
#define MM(M,y)   M = # y
M(M)(A,B)
```

6. 编写一系列的预处理器指令，如果SIZE宏没有被定义或者虽然被定义但它的值不在1-10的范围之内，它们将导致标准C程序无法通过编译。
7. 提供一个字符序列的例子，它可以作为一个单独的标记提供给预处理器，但它对于编译器却是不正确的。
8. 下面这个程序片段有什么问题？

```
if (x != 0)
   y = z/x;
else
   # error "Attempt to divide by zero, line " __LINE__
```

第4章 声 明



在C语言中，声明一个名称就是把一个标识符与一个C对象（例如变量、函数或类型）相关联。在C中，可以被声明的名称包括：

- 变量。
- 函数。
- 类型。
- 类型标签。
- 结构和联合的成员。
- 枚举常量。
- 语句标签。
- 预处理器宏。

除了语句标签和预处理器宏之外，所有的标识符都出现在C的*声明*（declaration）中。变量、函数和类型出现在声明的*声明器*（declarator）中，类型标签、结构和联合成员、枚举常量出现在声明中特定类型的*类型指定符*（type specifier）中。语句标签是在C函数中声明的，预处理器宏是由#define预处理器命令声明的。

由于一些原因，很难对C的声明进行描述。首先，它们涉及一些不同寻常的语法，可能会令初学者感到困惑。例如，下面这个声明

```
int (*f)(void);
```

它声明了一个函数指针，它所指向的函数不接受任何参数，并且返回一个整数。

其次，声明的许多抽象属性，例如作用域和范围，在C语言中比其他编程语言更为复杂。在讨论实际的声明语法之前，我们将在第4.2节讨论这些属性。

最后，如果对C的类型系统没有很好的了解，很难理解C语言声明的某些方面的特性，第5章将描述C的类型系统。对类型标签、结构和联合成员、枚举常量的讨论将推迟到那一章。不过为了完整起见，本章也讨论了这些声明的一些属性。

参考：*枚举类型* 5.5；#define预处理器命令 3.3；*语句标签* 8.3；*结构类型* 5.6；*类型指定符* 4.4；*联合类型*5.7。

4.1 声明的组织形式

在C程序中，声明可以出现在几个地方。声明所出现的位置将会影响它们的属性。C源文件又称*翻译单元*（translation unit），是由一系列的顶层函数、变量和其他对象声明所组成的。每个函数具有*参数声明*（parameter delcaration）和*函数体*，函数体又可以包含各种代码块，包括复合语句。代码块可以包含一系列的*内层声明*（inner declaration）。

下面显示了声明的基本语法。对函数定义的讨论将推迟到第9章。

声明：

声明指定符　初始化声明器列表；

声明指定符：

存储类别指定符　声明指定符$_{opt}$

类型指定符 声明指定符$_{opt}$
类型限定符 声明指定符$_{opt}$
函数指定符 声明指定符$_{opt}$ (C99)

初始化声明列表：

初始化声明器
初始化声明器列表，初始化声明器

初始化声明器：

声明器
声明器 = 初始化列表

在声明指定符中，最多只能出现1个存储类别指定符和1个类型指定符，但一个类型指定符可以由几个标记组成（例如，unsigned long int）。C99规定必须使用类型指定符。在声明指定符中，每个类型限定符最多只能出现1次。C99函数指定符（inline）可以出现在函数声明中。在这些限制中，类型指定符、存储类别指定符、函数指定符和类型限定符在声明指定符中可以按照任何顺序出现。

例子

习惯的做法是把存储类别指定符放在最前面，然后是任何类型限定符，最后是类型指定符。在下面这些声明中，i和j具有相同的类型和存储类别，但i的声明风格更好。

```
unsigned volatile long extern int const j;
extern const volatile unsigned long int i;
```

参考：声明器 第4.5节；表达式 第7章；函数定义 第9章；初始化列表 第4.6节；语句 第8章；存储类别指定符 第4.3节；类型指定符和限定符 第4.4节。

4.2 术 语

本节介绍用于描述声明的一些术语。

4.2.1 作用域

声明的作用域就是该声明在C程序文本中可见的那部分区域。在C中，标识符可以是表4-1所列出的6种类型之一。

表4-1 标识符的作用域

类 型	声明的可见性
顶层标识符	从它的声明位置（第4.2.3节）扩展到源程序文件的末尾
函数定义中的形式参数	从它的声明位置扩展到函数体的末尾
函数原型中的形式参数[a]	从它的声明位置扩展到原型的末尾
代码块标识符	从它在代码块中的声明位置扩展到代码块的末尾
语句标签	包括它所在的整个函数体
预处理器宏	从声明它的#define命令扩展到源程序文件的末尾，或者遇到取消它的定义的第1条#undef命令

a 为标准C所新增。

在函数定义或代码块中声明的非预处理器标识符（包括形式参数）常常被称为具有代码块作用域或局部作用域。原型中的标识符具有原型作用域。语句标签具有函数作用域。所有其他标识符具有文件作用域。

代码块一般是条复合语句。在C99中，还存在与选择和迭代语句有关的隐式代码块。

每个标识符的作用域限制在它所在的C源文件。但是，有些标识符可以声明为external，同一个标识符可以出现在2个或更多的文件中，并通过第4.8节所描述的方法进行链接。

参考：#define预处理器命令 第3.3节；外部名称 第4.8节；原型 第9.2节；#undef预处理器命令 第3.3节。

4.2.2 可见性

在某个上下文环境中，如果在使用某个标识符时将会绑定到该标识符的某个声明（即这个标识符与这个声明相关联），那么这个声明在这个上下文环境中就是可见的。声明在它自己的作用域内是可见的，但可能会被作用域和可见性与自己重叠的其他声明所隐藏。

例子

在下面这个程序中，foo的外层声明（整型变量）被内层声明（浮点型变量）所隐藏。外层的foo只在main函数体内被隐藏。

```
int foo = 10; /* foo defined at the top level */
int main(void)
{
    float foo; /* this foo hides the outer foo */
    …
}
```

在C中，位于代码块开始位置的声明可以隐藏这个代码块外部的声明。为了让一个声明隐藏另一个声明，它们所声明的标识符必须相同、必须属于相同的重载分类（overloading class）、必须在两个不同的作用域中声明，并且其中一个标识符的作用域嵌套于另一个标识符的作用域的内部。

在标准C中，函数定义中形式参数声明的作用域与在表示函数体的代码块的开始位置所声明的标识符相同。但是，有些早期的C编译器使用了参数作用域，涵盖了代码块作用域。

例子

下面x的重新声明在标准C中是错误的，但有些旧式的编译器允许这种做法，这可能会导致难以检测的编程错误。

```
int f(x)
    int x;
{
    long x = 34;      /* invalid? */
    return x;
}
```

参考：代码块 第8.4节；重载分类 第4.2.4节；参数声明 第9.3节；顶层声明 第4.1节。

4.2.3 前向引用

一般情况下，标识符在完整声明之前可能无法被正确地使用。为了准确起见，我们把标识

符的声明点（declaration point）定义为包含这个标识符的词法标记的声明器的最后。在声明点之后使用标识符是允许的。在下面这个例子中，整数变量intsize可以被初始化为它自己的长度，因为在初始化值中所使用的intsize出现在声明点的后面：

```
static int intsize = sizeof(intsize);
```

当一个标识符在完整声明之前被使用时，就出现了该声明的前向引用（forward reference）。C允许在下面这三种情况下使用前向引用：

1. 语句标签可以提前出现在goto语句中，因为它的作用域覆盖整个函数体。

```
if (error) goto recover;
…
recover:
   CloseFiles();
```

2. 可以声明不完整的结构、联合、数组或枚举类型，以便在完整定义之前使用它们，以实现某些目的。

3. 函数的声明可以与它的定义分离，它可以是显式的函数声明，或者隐式地出现在函数调用中（第4.7节和第5.8节）。C99不允许通过函数调用隐式地声明一个函数。

例子

这个例子说明了非法的前向声明。程序员试图用typedef声明定义一个自引用的结构。在这个例子中，cell在该行代码中最后出现的位置才是声明点，因此在这个结构中使用cell是非法的。

```
typedef struct { int Value; cell *Next; } cell;
```

声明这种类型的正确方法是使用结构标签S，它是在第一次出现时被定义的，因此可以在声明中使用它：

```
typedef struct S { int Value; struct S *Next; } cell;
```

另外，可以参见后面对隐式声明（第4.7节）和重复声明（第4.2.5节）的讨论。

参考：重复声明 第4.2.5节；函数类型 第5.8节；goto语句 第8.10节；隐式声明 第4.7节；指针类型 第5.3节；结构类型 第5.6节。

4.2.4 名称的重载

在C和其他编程语言中，一个标识符可以同时与多个程序实体相关联。如果出现了这种情况，我们就认为这个名称被“重载”，这个名称在使用时的上下文环境决定了与它相关联的是哪个实体。例如，一个变量和一个结构标签可能使用相同的标识符。当这个标识符在一个表达式中使用时，与它相关联的就是那个变量。当这个标识符在一个类型指定符中使用时，与它相关联的就是那个结构标签。

在C中，名称并有5种重载分类（有时候把它们称为名字空间）。表4-2列出了这些重载分类，并对它们进行了描述。

表4-2 重载分类

分 类	所包含的标识符
预处理器宏名	由于预处理逻辑发生在编译之前，预处理器所使用的名称与C程序的其他所有名称独立

（续）

分 类	所包含的标识符
语句标签	有名称的语句标签是语句的一部分。在语句标签的定义中，名称后面总是有:（它并不是case标签的一部分）。在使用语句标签时，它总是出现在关键字goto的后面
结构、联合和枚举标签	这些标签是结构、联合和枚举类型指定符的一部分。如果存在，它们总是出现在关键字struct、union或enum的后面
成分名称（标准C中的“成员”）	成分名称是在与每个结构和联合类型相关联的名字空间中分配的。也就是说，同一个标识符可以同时是任何数量的结构或联合的成分名称。成分名称的定义总是出现在结构或联合类型指定符的内部。在使用成分名称时，它们总是出现在选择操作符.或->的后面
其他名称	所有其他名称属于一种重载分类，包括变量、函数、typedef名称和枚举常量

这些重载规则与C最初所定义的重载规则稍有不同。首先，语句标签最初和普通的标识符属于同一个名字空间。其次，所有的结构和联合成员名称属于一个名字空间，而不是像原来那样分属各自的名字空间。

当一个名称进行了重载，并与几个实体相关联时，每个关联具有它自己的作用域，并可能被其他声明所隐藏（但与其他关联无关）。例如，如果一个标识符既作用一个变量的名称又作为一个结构标签，一个内层代码块可能会重新定义变量的关联，而不会更改结构标签的关联。

C++把结构和联合标签归入“其他”名字空间（第4.9.2节）。

参考：成分名称 第5.6.3节；重复定义 第4.2.5节；枚举标签 第5.5节；goto语句 第8.10节；选择操作符 第7.4.2节；语句标签 第8.10节；结构标签 第5.6节；结构类型指定符 第5.6节；typedef名称 第5.7节；联合类型定符 第5.7节。

4.2.5 重复声明

在同一个代码块或者在顶层对同一个名称进行两次声明（属于相同的重载分类）是非法的。这种定义被认为出现了“冲突”。

例子

在下面这个例子中，howmany的两个声明出现了冲突，但str的两个声明并没有冲突，因为它们位于不同的名字空间。

```
extern int howmany;
extern char str[10];
typedef double howmany();
extern struct str {int a, b;} x;
```

重复声明在两种情况下是合法的。首先，相同名称的外部（引用）声明可以出现任意次数，只要它们在每种情况下都把正确的类型赋值给这个名称。这个例外反映了一个情况，就是多次声明同一个外部库函数并不是非法的。

其次，如果一个标识符进行了外部声明，这个声明的后面可以出现它的定义（第4.8节），只要这个定义和外部声明把相同的类型赋值给这个标识符。这个例子允许用户对变量和函数进行前向引用。

例子

我们定义了两个函数f和g，它们相互引用。正常情况下，在g中使用f将是非法的前向引用。但是，在g的定义中添加f的外部声明，使编译器在编译g时能够知道和f有关的足够信息（如果没有f的初始声明，那些只执行一道编译的编译器在编译g时就无法知道f返回一个double类型的值）。

```
extern double f(double z);

double g(double x, double y)
{
    … f(x-y) …
}

double f(double z)
{
    … g(z, z/2.0) …
}
```

参考：定义和引用声明 第4.8节；extern存储类别 第4.3节；前向引用 第4.2节；重载分类 第4.2节；static存储类别 第4.3节。

4.2.6 重复可见性

由于C的作用域规则指定了一个名称的作用域从它的声明点开始，而不是从定义它的那个代码块的头部开始，这就可能产生一种情况，就是可以在同一个代码块的不同部分引用两个不冲突的声明。

例子

在下面这段代码中，在标签为B的代码块中引用了两个称为i的变量。整数i是在外层代码块中声明的，用于初始化变量j。接着，代码块B声明了浮点变量i，从而隐藏了第1个i。

```
{
    int i = 0;
    …
    B: {
            int j = i;
            float i = 10.0;
            …
        }
}
```

在j进行初始化时，对i的引用存在歧义。哪一个i是它所需要的呢？大多数编译器会执行我们所预想的操作，代码块B对i的第1个使用将绑定到它的外层定义。对i的重定义将在代码块B的剩余部分隐藏i的外层定义。这是标准C所采用的规则。我们把这种做法看成是不良的编程风格，应该尽量避免。

4.2.7 范围

与类型不同，变量和函数在运行时是存在的，也就是说，编译器为它们分配了存储空间。这些对象的范围（或生命期）就是这些存储空间在分配之后所存在的时间。标准C称之为*存储持久性*（storage duration）。

如果一个对象在程序执行之前就分配了存储空间，并且在程序终止之前这个存储空间一直有效，这个对象就被认为具有*静态范围*（static extent）。在C中，所有的函数都具有静态范围，

所有在顶层声明的变量也是如此。在代码块内部声明的变量也可能具有静态范围，这取决于它们的声明方式。

如果一个对象是在一个代码块或函数的开始处声明的，并且当这个代码块或函数退出时它将被销毁，这个对象就被认为具有*局部范围*（local extent）。如果一个具有局部范围的变量在声明时具有初始值，则这个变量在每次创建时都会进行初始化。形式参数具有局部范围，在代码块的起始处声明的变量也可能具有局部范围，也是取决于它们的声明方式。在C中，具有局部范围的变量又称自动变量。

最后，C还存在具有*动态范围*（dynamic extent）的对象。这些对象的创建和销毁是根据程序员的思路显式地进行的。但是，动态对象必须使用特殊的库函数（例如malloc）创建，而这些库函数并不被认为是C语言的一部分。

参考：auto*存储类别* 第4.3节；*初始化值* 第4.6节；malloc*函数* 第16.1节；static*存储类别* 第4.3节；*存储分配函数* 第16.1节。

4.2.8 初始值

为一个变量分配存储空间时，并不一定要创建这个存储空间的初始内容。C的大多数声明可以具有初始化值，也就是在为变量分配存储空间时用于设置变量初始值的表达式。如果没有为局部变量指定初始化值，则这个局部变量的值在分配存储空间之后是无法预测的（静态变量被默认初始化为零）。

注意，静态变量只初始化一次，即使程序的执行离开了它的作用域，它仍然保持原来的值。

例子

在下面的代码中，两个变量L和S是在一个代码块的起始位置声明的，它们都被初始化为0。这两个变量都具有局部作用域，但S具有静态范围，L具有局部（自动）范围。每次进入这个代码块时，这两个变量的值都将增加1，并打印出它们的新值。

```
{
    static int S = 0;
    auto int L = 0;
    L = L + 1;
    S = S + 1;
    printf("L = %d, S = %d\n", L, S);
}
```

哪些值将被打印出来呢？如果这个代码块被执行许多次，输出结果将是：

```
L = 1, S = 1
L = 1, S = 2
L = 1, S = 3
L = 1, S = 4
...
```

对于在代码块的起始位置所声明的自动变量，C的初始化具有一个危险的特性。只有当代码块是通过正常方式进入时（即控制流进入代码块的起始处），才会保证执行初始化。通过语句标签和goto语句，可以直接跳到一个代码块的中间位置。如果采用这种做法，就无法保证自动变量被初始化。事实上，大多数标准和非标准的编译器并不会对它们进行初始化。在switch语句中，

控制跳转到一个作为case或default标签语句体的代码块是很正常的。因此，在第1个这类标签之前的自动变量将不会被初始化。

例子

当goto语句把控制转移到标签L时，变量sum很可能没有进行初始化。这将导致sum的值是不确定的。

```
goto L;
…
{
    static int vector[10] = {1,2,3,4,5,6,7,8,9,10};
    int sum = 0;
L:
    /* Add up elements of "vector". */
    for ( i=0; i<10; i++ ) sum += vector[i];
         printf("sum is %d", sum);
}
```

参考：goto语句 第8.10节；变量的初始化 第4.6节；存储类别 第4.3节；switch语句 第8.7节。

4.2.9 外部名称

作用域和可见性的一种特殊情况就是外部标识符，又称为具有外部链接属性的标识符。在组成C程序的所有文件中，一个外部标识符的所有实例必须引用同一个对象或函数，并且必须在每个文件中被声明为兼容的类型，否则它的结果就是未定义的。

外部名称必须显式或隐式地用external关键字进行声明。但是，并不是所有声明为external的名称都是外部名称。外部名称通常是在C程序的顶层声明的，因此具有文件作用域。但是，非标准的编译器在处理在代码块内部所声明的外部名称时可能采用不同的行为。

例子

下面这个程序片段在许多C编译器中都是可以接受的。它在一个代码块内部声明了一个外部名称，并在这个代码块的外面使用这个名称：

```
{
    extern int E;
    …
}
E = 1;
```

根据正常的代码块作用域规则，这个声明在代码块之外是不可见的。但是，许多C编译器隐式地为E设置了文件作用域，因此在编译这段代码时不会出错。标准C要求这个声明具有代码块作用域，但并没有表示像前面这样的代码段应该是非法的。从技术上说，编译器在这种情况下的行为是未定义的，因此允许遵循C标准的编译器接受这个程序。我们认为程序员应该把这样的程序片段看成是个程序错误，即使编译器可以接受它并且它的运行时结果也是正确的。

毫无疑问，如果两个外部声明（位于同一个文件或位于同一个程序的不同文件）为同一个标识符指定了不兼容的类型，将是一种错误的行为。

例子

在下面这个程序中，X的两个声明在源文件中并没有冲突，但它们的行为在运行时是未定义的：

```
int f() { extern int X; return X; }
double g() { extern double X; return X; }
```

参考：外部名称约定 第2.5节；外部名称定义和引用 第4.8节；作用域 第4.2.1节；类型兼容 第5.11节；可见性 第4.2.2节。

4.2.10 编译时名称

到目前为止，我们的讨论主要集中在变量和函数，它们在运行时存在。但是，作用域和可见性规则普遍适用于与对象相关联的标识符，这些对象在运行时并不一定存在，例如typedef名称、类型标签和枚举常量。在声明这类标识符时，它们的作用域与在这个位置所声明的变量相同。宏和标签也是编译时名称，但它们的作用域是不同的。

参考：枚举常量 第5.5节；作用域 第4.2.1节；结构类型 第5.6节；typedef名称 第5.10节；可见性 第4.2.2节。

4.3 存储类别和函数指定符

现在，我们继续讨论声明的组成部分：存储类别指定符、类型指定符、类型限定符、函数指定符、声明器和初始化值。

存储类别指定符决定了被声明对象的范围（除了typedef，它属于特殊情况）。在一个声明中，最多只能出现1个存储类别指定符。在声明中，把存储类别指定符放在类型指定符和类型限定符之前是习惯的做法。

存储类别指定符：下列之一

```
auto  extern  register  static  typedef
```

表4-3列出了这些存储类别的含义。注意，并不是所有的存储类别都可以出现在任何声明上下文环境中。

表4-3 存储类别指定符

指定符	用 途
auto	只允许在代码块内部[a]变量的声明中使用。它表示变量具有局部（自动）范围（由于这是默认设置，所以C程序中很少看到auto的显式用法）
extern	可以出现在外部函数和变量的声明中，声明可以位于顶层或者位于代码块的内部[a]。它表示对象声明具有静态范围，它的名称是链接器所知的。参见第4.8节
register	可以用于局部变量和形参声明。它相当于auto，区别在于它向编译器提示这个变量将被大量使用，在分配存储时应该把它放在访问时间最少的地方
static	可以出现在函数或变量的声明中。在函数定义中，它只用于指定函数名不被导出到链接器。在函数声明中，它表示被声明的函数稍后将在这个文件中按照static存储类别进行定义。在数据声明中，它总是表示一个不导出到链接器的定义声明。声明为这个存储类别的变量具有静态范围（和auto所指定的局部范围正好相反）
typedef	表示这个声明是在为一种数据类型（而不是变量或函数）定义一个新名称。数据类型的名称出现在变量声明中变量名称所出现的位置，数据类型本身是将被赋值给变量名的类型（参见第5.10节）

a C99允许声明出现在代码块中的任何位置。以前版本的C只允许声明出现在代码块的第1条语句之前。

标准C允许在任何类型的变量或形参中使用register，但是不允许获取这种对象的地址，不管

是显式（使用&操作符）还是隐式（例如，对数组进行下标操作时把数组名转换为指针）。许多非标准的C编译器具有不同的行为：

- 它们可能会施加限制，只允许对标量类型的对象使用register。
- 它们可能允许在register对象上使用&操作符。
- 它们可能会隐式地加宽用register声明的较小对象（例如，把声明register char x看成是register int x）。

编译器可以把存储类别指定符register看成与auto相同。但是，程序员可以在函数中对1到2个大量使用的变量使用register，以提高它们的性能。对许多声明同时使用register反而会产生副作用，导致效率低下。对于最为现代的编译器而言，使用register的效果并不显著，因为这些编译器很可能已经根据程序的需要把使用最为频繁的变量分配到寄存器。

参考：取地址操作符& 第7.5.6节；形式参数声明 第9.3节；初始化值 第4.6节；下标 第7.4.1节；顶层声明 第4.1节；typedef名称 第5.10节。

4.3.1 默认存储类别指定符

如果在声明中并未提供存储类别指定符，这个声明将根据表4-4所显示的声明上下文环境来确定它的存储类别。

表4-4 默认存储类别指定符

声明的位置	声明的类型	默认存储类别
顶层	全部	extern
函数的形参	全部	无（即“非register”）
在代码块内部	函数	extern
在代码块内部	非函数	auto

在顶层声明中省略存储类别指定符的效果并不一定等于extern，原因如第4.8节所述。作为一种良好的编程风格，我们认为程序员在代码块内部声明外部函数时应该加上存储类别extern。C程序中极少看到auto关键字，它一般是默认值。

参考：代码块 第8.4节；形参声明 第9.3节；顶层声明 第4.1节、第4.8节。

4.3.2 存储类别指定符的例子

下面显示了堆排序算法的一个实现，它的具体细节超出了本书的范围。

例子

这个算法把一个数组当成是一棵二叉树，元素b[k]的两棵子树分别是元素b[2*k]和b[2*k + 1]。这里所说的堆，是指每个节点所包含的数都不小于它的所有子节点之数的树。

```
#define SWAP(x, y) (temp = (x), (x) = (y), (y) = temp)

static void adjust (int v[], int m, register int n)
/* If v[m+1] through v[n] is already in heap form,
   this puts v[m] through v[n] into heap form. */
{
   register int *b, j, k, temp;
   b = v - 1; /* b is "1-origin", customary in heapsort,
                 i.e., v[j] is the same as b[j-1] */
```

```
    j = m;
    k = m * 2;
    while (k <= n) {
        if (k < n && b[k] < b[k+1]) ++k;
        if (b[j] < b[k]) SWAP(b[j], b[k]);
        j = k;
        k *= 2;
    }
}

/* Sort v[0]…v[n-1] into increasing order. */
void heapsort(int v[], int n)
{
    int *b, j, temp;
    b = v - 1;
    /* Put the array into the form of a heap. */
    for (j = n/2; j > 0; j--) adjust(v, j, n);
    /* Repeatedly extract the largest element and
       put it at the end of the unsorted region. */
    for (j = n-1; j > 0; j--) {
        SWAP(b[1], b[j+1]);
        adjust(v, 1, j);
    }
}
```

辅助函数adjust并不需要在外部可见，因此它被声明为static。adjust函数的速度对于排序的效率是极为关键的，因此它的局部变量被声明为register，提示编译器对它们进行优化。形式参数n在adjust中也被反复使用，因此也用存储类别指定符register指定。adjust的另两个形式参数默认为“非register”。

主函数是heapsort，它必须对这个排序程序包的用户可见，因此使用默认的存储类型指定符extern。heapsort函数的局部变量对性能并没有太大影响，因此使用默认的存储类别auto。

4.3.3 函数指定符

函数指定符是C99所新增的。

函数指定符：

```
inline                                    (C99)
```

函数指定符inline只能出现在函数声明中，这种函数便成为内联函数。这个指定符可以多次出现，而不会改变含义。使用inline是向编译器所提供的提示，表示应该采用措施提高这个函数的调用速度。

第9章将讨论内联函数的详细规则。

参考：内联函数 第9.10节。

4.4 类型指定符和限定符

类型指定符提供了和被声明的程序标识符的数据类型有关的信息。其他的类型信息是由声明器所提供的。类型指定符也可以定义（作为副作用）类型标签、结构和联合成员名以及枚举常量。

类型限定符const、volatile和restrict指定了类型的其他属性。只有当通过左值访问这些类型

的对象时，这些限定符才会发生作用。

类型指定符：

枚举类型指定符
浮点类型指定符
整数类型指定符
结构类型指定符
typedef名称
联合类型指定符
void类型指定符

类型限定符：

```
const
volatile
restrict                                    (C99)
```

例子

下面是类型指定符的一些例子：

```
void                    union { int a; char b; }
int                     enum {red, blue, green}
unsigned long int       char
my_struct_type          float
```

第5章将详细描述类型指定符，因此对特定的类型指定符的讨论也将推迟到第5章。但是，接下来各节讨论了和类型指定符有关的一些基本问题。

参考：声明器 第4.5节；枚举类型指定符 第5.5节；浮点类型指定符 第5.2节；整数类型指定符 第5.1节；左值 第7.1节；结构类型指定符 第5.6节；类型限定符 第4.4.3节；typedef名称 第5.10节；联合类型指定符 第5.7节；void类型指定符 第5.9节。

4.4.1 默认类型指定符

最初，C允许在变量声明和函数声明中省略类型指定符。在这种情况下，类型指定符默认为int。在现代C中，这被认为是一种不良的编程风格，C99更是规定这种做法是错误的。旧式的编译器并没有实现void类型，因此在函数定义中省略类型指定符的基本原理是提醒读者这类函数实际上并没有返回值（尽管编译器认为它返回了一个值）。

例子

在标准C之前，下面这种形式的函数定义很常见：

```
/* Sort v[0]…v[n-1] into increasing order. */
sort(v, n)
    int v[], n;
{
    …
}
```

在现代标准C风格的编译器中，一般把这类函数声明为void类型：

```
/* Sort v[0]…v[n-1] into increasing order. */
```

```
void sort(int v[], int n)
{
    …
}
```

例子

在使用并没有实现void的编译器时，自己定义void并显式地在程序中使用显然要比完全省略类型指定符好得多：

```
/* Make "void" be a synonym for "int". */
typedef int void;
```

就我们所知道的旧式编译器而言，至少有一个保留了标识符void，但并没有实现它。对于这种编译器，下面这条预处理指令

```
#define void int
```

是少数几个使用宏名代替关键字被认为合理的地方。

例子

声明的语法（第4.1节）要求声明包含1个存储类别指定符、1个类型指定符、1个类型限定符或它们的其他组合。这个需求避免了语言的语法歧义。如果所有的指定符和限定符均省略，下面这个声明

```
extern int f();
```

就简单地成为

```
f();
```

它在语法上等于一条包含了一个函数调用的语句。我们认为，最好的风格是始终包含类型指定符，并允许存储类别指定符为默认值，尤其是当它为auto的时候。

例子

对LALR（1）语法着迷者的最后一点说明：存储类别指定符和类型指定符均可在函数定义中省略，这在C程序中是极为常见的，例如：

```
main() { … }
```

在这种情况下，并不会产生语法上的歧义，因为函数声明中的声明器后面必须是逗号或冒号，而函数定义中的声明器后面必须是个左花括号。

参考：声明 第4.1节；函数定义 第9.1节；void类型指定符 第5.9节。

4.4.2 缺失的声明器

下面的讨论处理声明和类型指定符的一个微妙之处。属于结构、联合或枚举定义的类型指定符定义了新的类型或枚举常量。如果想简单地定义一种类型，在声明中省略所有的声明器、只书写类型指定符也是合理的。在标准C中，声明必须包含一个声明器，定义一个结构或联合标签，或者定义枚举常量。在传统C中，常常会静静地忽略不合理的声明。

例子

下面这个声明由一个类型指定符组成。它定义了一个新的结构类型S，后者包含成员a和b。

```
struct S { int a, b; }; /* Define struct S */
```

以后，这种类型就可以只用这个指定符来表示

```
struct S x, y, z; /* Define 3 variables */
```

但是，下面的声明则不合理，并且在标准C中是非法的：

```
struct { int a, b; };              /* no tag */
int ;                              /* no declarator */
static struct T { int a, b; }; /* extra storage class */
```

在第1个声明中，由于缺少结构标签，所以将来就无法在这个程序中引用这种类型。在第2个声明中，它根本就没有效果。第3个声明提供了一个存储类别指定符，但它将被忽略。读者可以会认为以后像下面这样的声明

```
struct T x, y;
```

将导致x和y具有存储类别static，但事实上并非如此。

参考：枚举类型 第5.5节；声明器 第4.5节；结构类型 第5.6节；类型指定符 第4.4节；联合类型 第5.7节。

4.4.3 类型限定符

C89增加了类型限定符const和volatile，C99增加了restrict。使用这些限定符的任意组合所声明的标识符被认为是限定类型，因此每个非限定版本具有7个可能的限定版本（限定符的顺序没有影响）。7种限定类型互不兼容，并且与对应的非限定版本也不兼容。如果同一个限定符在同一个声明中多次出现，C99将会忽略多余的限定符，C89则把它当成是一个错误。

类型限定符指定了类型的额外属性，只有当通过左值（指示符）访问具有限定符的对象时，这些限定符才会发挥作用。当限定符在需要一个值而不是一个指示符的上下文环境中使用时，该限定符就不会对类型产生任何影响。也就是说，在表达式L=R中，=操作符的右操作数的类型总是非限定版本的，即使它在声明时使用了类型限定符。但是，左操作数仍然保持它的限定性，因为它是作为左值使用的。

除了出现在顶层声明中，类型限定符还可以出现在指针声明器中，甚至出现在数组声明器中（C99）。

例子

使用不支持类型限定符的C编译器时，可以提供下面的宏定义，这样程序中即使出现了类型限定符也不会导致编译失败。当然，在这种情况下，限定符没有任何效果。

```
#ifndef __STDC__
#define const /*nothing*/
#define volatile /*nothing*/
#define restrict /*nothing*/
#endif
```

参考：#ifndef 第3.5.3节；__STDC__ 第11.3节；数组声明器 第4.5.3节；指针声明器 第4.5.2节；类型兼容性 第5.11节。

4.4.4 const

具有const限定类型的左值表达式的值无法被修改。也就是说，这种左值不能作为赋值表达式的左操作数，也不能作为增值和减值操作符的操作数。使用const限定符的意图是表示值不会

变化的对象，并让C编译器保证程序员并不会修改这种左值的值。

例子

下面的声明指定了ic是个具有常量值37的整数：

```
const int ic = 37;
…
ic = 5; /* Invalid */
ic++;   /* Invalid */
```

const限定符还可以出现在指针声明器中，把指针声明为“常量指针”和“指向常量数据的指针”：

```
int * const const_pointer;
const int *pointer_to_const;
```

这种语法比较容易混淆。例如，常量指针和常量整数在不同的位置具有类型限定符const。在使用typedef名称时，const限定符的用法也会有所变化。在前面这个例子中，常量指针const_pointer可以会按照下面的方式声明：

```
typedef int *int_pointer;
const int_pointer const_pointer;
```

这使得const_pointer看上去就像“指向常量int_pointer的指针”，但事实上并不是这样，它仍然是个指向（非常量）int的常量指针。事实上，由于类型指定符和类型限定符的顺序并不会产生影响，所以最后一个声明也可以写成：

```
int_pointer const const_pointer;
```

可以把变量的类型修改为“指向常量数据的指针”，但它所指向的对象无法被修改。对const限定的类型的值应用取地址操作符&可以产生这种类型的表达式。为了保护常量数据的完整性，必须使用显式的类型转换，才允许把“指向const T的指针”类型的值转换为“指向T的指针”类型的对象。

例子

```
const int *pc; /* pointer to a constant integer */
int *p, i;
const int ic;
pc = p = &i;     /* OK */
pc = &ic;        /* OK */
*p = 5;          /* OK */
*pc = 5;         /* Invalid */
pc = &i;         /* OK */
pc = p;          /* OK */
p = &ic;         /* Invalid */
p = pc;          /* Invalid */
p = (int *)&ic; /* OK */
p = (int *)pc;  /* OK */
```

const的语言规则并不是很安全。也就是说，如果程序员有心，可以绕过或者重写它们。例如，常量对象的地址可以传递给一个没有原型的外部函数，这个函数可以修改这个常量对象。但是，编译器可以在只读存储区中为const限定类型的静态对象分配内存，这样试图修改这种对

象将会导致运行时错误。

例子

下面这个程序片段说明了绕过const限定符而导致的危险。

```
const int * pc;
int * p;
const int ic = 0;
…
pc = &ic;        /* OK */
p = (int *)pc;   /* Valid, but dangerous */
*p = 5;          /* Valid, but may cause a run-time error */
```

最后，具有类型限定符const并且没有显式声明存储类别的顶层声明被认为具有extern存储类别。

参考：赋值表达式 第7.9节；增值和减值表达式 第7.4节；指针声明器 第4.5.2节。

4.4.5 volatile和序列点

volatile类型限定符告诉标准C编译器，有些对象的值可能不受编译器控制而发生改变。volatile对象（即通过使用volatile限定符的左值表达式所访问的任何对象）不应该参与必须没有隐藏副作用才能进行的优化。

更准确地说，标准C在C程序中引入了序列点（sequence point）的概念。序列点存在于当所有的表达式（不是更大的表达式的一部分）都完成时，也就是在表达式语句结束时；在if、switch、while和do语句的控制表达式之后；在for语句的3个控制表达式的每个之后；在逻辑AND（&&）、逻辑OR（||）、条件操作符（?:）、逗号操作符（,）的第1个操作数之后；在return语句的表达式之后以及在初始化值之后。其他的序列点还包括在完整声明器的最后、在函数调用中所有的实参都进行了求值之后、在库函数返回之前、在与printf/scanf转换指定符相关联的操作完成之后，以及在调用bearch和qsort的比较函数的前后。

对volatile对象的引用和修改进行优化时，不能跨越序列点，尽管序列点之间的优化是允许的。除此之外，C语言标准允许在源代码的其他地方对它们进行引用和修改。但是，根据我们的经验，程序员倾向于编译器只以“写入”的方式对volatile对象进行访问和修改。程序员可以很方便地复制一个volatile对象的值，然后对它进行优化。

例子

考虑下面这个程序片段，其中j在循环之前被赋值为某个值：

```
extern int f(int);
auto int i,j;
…
i = f(0);
while (i) {
    if (f(j*j)) break;
}
```

如果变量i在它的生命期内并未再次使用，则传统的C编译器就可以把这个程序片段改写为：

```
if (f(0)) {
    i = j*j;
    while( !f(i) ) ;
}
```

对i的第1次赋值被消除，i作为一个临时变量被复用，保存j*j的值，后者只在循环之外求值1次。如果i和j的声明是下面这样：

```
auto volatile int i,j;
```

这种情况下就不允许对它们进行优化。但是，我们可以把这个循环写成下面这样，消除if语句的控制表达式尾部的序列点之前对j的一次引用：

```
i = f(0);
while (i) {
    register int temp = j;
    if (f(temp*temp)) break;
}
```

新的指针声明器语法允许声明“指向volatile…的指针”，对这种类型的指针的引用可以进行优化，但对它所指向的对象的引用却不能进行优化。只有在进行了显式的类型转换时，才允许把“指向volatile T的指针”类型的值赋值给“指向T的指针”类型的对象。

例子

下面是一些使用volatile对象的合法例子和非法例子：

```
volatile int * pv;
int *p;
pv = p;    .     /* OK */
p = pv;          /* Invalid */
p = (int *)pv; /* OK */
```

volatile最常见的应用是对由计算机硬件或异步进程（例如中断处理程序）所使用的特殊内存位置进行可靠的访问。

例子

考虑下面这个典型例子。一台计算机具有3个特殊的位置：

地址	使用
0xFFFFFF20	输入数据缓冲区
0xFFFFFF24	输出数据缓冲区
0xFFFFFF28	控制寄存器

控制寄存器和输入数据缓冲区可以由程序读取，但不能写入。输出数据缓冲区可以被写入，但不能被读取。控制寄存器的第3个最低位称为“输入可用”。当存在来自外部设备的数据时，这个位就被设置为1。当程序从输入缓冲区中读取了这些数据之后，这个位就被自动设置为0（在此之后，输入数据缓冲区的内容是未定义的，直到这个“输入可用”位再次被设置为1）。控制寄存器的第2个最低位称为“输出可用”。当外部设备打算接收数据时，这个位就被设置为1。当数据被程序放置在输出缓冲区中时，这个位就会被自动设置为0，并且数据被写出。当这个控制位为0时把数据放在输出缓冲区中会导致不可预测的结果。

copy_data函数接着把数据从输入复制到输出，直到它遇到输入值为0。这个函数返回它所复制的字符数量。它并没有采用任何措施防止溢出和其他错误条件。

```
typedef unsigned long datatype, controltype, counttype;

#define CONTROLLER \
```

```
    ((const volatile controltype * const) 0xFFFFFF28)
#define INPUT_BUF \
    ((const volatile datatype * const) 0xFFFFFF20)
#define OUTPUT_BUF \
    ((volatile datatype * const)  0xFFFFFF24)
#define input_ready ((*CONTROLLER) & 0x4)
#define output_ready ((*CONTROLLER) & 0x2)
counttype copy_data(void)
{
    counttype count = 0;
    datatype temp;
    for(;;) {
        while (!input_ready) ;     /* Wait for input */
        temp = *INPUT_BUF;
        if (temp == 0) return count;
        while (!output_ready) ;    /* Wait to do output */
        *OUTPUT_BUF = temp;
        count++;
    }
}
```

参考：bsearch 第20.5节；转换规范 第15.8.2节、第15.11.2节；声明器 第4.5节；初始化值 第4.6节；指针声明器 第4.5.2节；qsort 第20.5节。

4.4.6 restrict

C99新增了类型限定符restrict，它只能用于限定指向对象或不完整类型的指针，它的用途是向C编译器提供“无别名”提示。这意味着这个指针是此时访问它所指向的对象的唯一方式。违反这个假设会导致未定义的行为。“此时”这个词意味着在有些情况下，可以根据原先restirct所限定的指针在函数体或代码块内部创建别名，只要这些别名在函数体或代码块结束时将被消除。C99标准提供了restrict的精确数学定义。下面给出了一些常见的情况。

1. 使用restrict声明并且具有文件作用域的指针被认为是访问它所指向的对象的唯一方法。对于在运行时由malloc进行初始化的全局指针而言，这是适当的用法。

```
extern double * restrict ptr;
...
void initialize(void)
{
    ptr = my_malloc( sizeof(double) );
}
```

2. restrict限定的指针如果是个函数的形参，它被认为是在这个函数执行开始时访问它所指向的对象的唯一方法，其他不是通过这个参数所创建的指针无法修改这个对象。例如，memcpy函数（和memmove函数不同）要求它的源内存区域和目标内存区域不能重叠。在C99中，这个要求可以在函数原型中表示：

```
#include <string.h>
void *memcpy(
    void * restrict s1,
    const void * restrict s2,
    size_t n);
```

3. 两个restrict限定的指针，或者其中一个具有restrict限定而另一个没有restrict限定，可以通过它们访问同一个对象，只要这个对象在restrict限定的指针的生命期内没有被修改。例如，考虑下面这个函数，它对两个向量进行求值，把和存储在第3个向量中：

```
void add(int n, int * restrict dest,
    int * restrict op1, int * restrict op2)
{
    int i;
    for (i = 0; i < n; i++)
        dest[i] = op1[i] + op2[i];
}
```

如果a和b是两个长度为N的不相邻数组，则调用add(N, a, b, b)，使op1和op2都指向数组b是正确的做法。这是因为数组b绝不会被修改。当然，它取决于程序员对add函数的实现方式的了解程度。只看到过add函数原型的程序员无法知道这种调用是安全的。

4. 结构的成员可以是restrict限定的指针。它的含义是：当一个结构实例被创建时，这个restrict限定的指针是访问它所指向的对象的唯一方法。

在restrict添加到C语言之前，程序员不得不依赖非移植性的pragma指令或编译器开关来启用这种类型的指针优化。如果在某一时刻只有一个指针能够访问一个对象，这种优化就是安全的。这种优化可以在运行时极大地提升速度。

省略restrict并不会改变程序的含义。C编译器可以忽略restrict。在本书中，许多库函数原型在编写时使用了restrict限定符。使用C99之前的编译器的程序员应该省略或忽略restrict。

参考：malloc 第16.1节；memcpy 第14.3节。

4.5 声 明 器

声明器引入了被声明的名称，并提供了其他的类型信息。以前的编程语言都没有提供和C的声明器相似的语法结构。

声明器：

指针声明器
直接声明器

直接声明器：

简单声明器
(声明器)
函数声明器
数组声明器

接下来各节描述了各种不同的声明器。

4.5.1 简单声明器

简单声明器用于定义算术、枚举、结构和联合类型的变量：

简单声明器：

标识符

假设S是个类型指定符，id是任何标识符。下面这个声明：

```
S id ;
```

表示id是S类型的。id被称为简单声明器。

例子

声明	x的类型
int x;	整数
float x;	浮点数
struct S { int a; float b; } x;	包含两个成员的结构

当类型指定符提供了所有的类型信息之后，就可以在声明中使用简单声明器。它适用于算术、结构、联合、枚举和void类型，以及由typedef名称所表示的类型。指针、数组和函数类型需要使用更复杂的声明器。但是，每个声明器都包含了一个标识符，因此我们可以认为声明器“封装”了一个标识符。

参考：类型指定符 第4.4节；结构类型 第5.6节；typedef名称 第5.10节。

4.5.2 指针声明器

指针声明器用于声明指针类型的变量。下面语法中的“类型限定符列表”是标准C新增的，它在旧式的编译器中被省略：

指针声明器：

指针 直接声明器

指针：

* 类型限定符列表$_{opt}$

* 类型限定符列表$_{opt}$ 指针

类型限定符列表： (C89)

类型限定符

类型限定符列表 类型限定符

假设D是封装了标识符id的任何声明器，并且“S D”这个声明表示id的类型是“…S”。则下面这个声明：

```
S *D ;
```

表示id的类型是“…指向S的指针”。指针声明器中可选的类型限定符列表只在标准C中是允许的。如果出现了类型限定符列表，这些限定符作用于指针，而不是作用于指针所指向的对象。

例子

在下面这张表格中x的三个声明器，id是x，S是int，“…”分别是“”、“的数组”和“返回…的函数”（这些很容易弄清楚，但解释起来很麻烦）。

声明	x的类型
int *x;	指向int的指针
int *x[];	指向int的指针数组
int *x();	返回一个指向int的指针的函数

例子

在下面的声明中，ptr_to_const是个指向常量int的（非常量）指针，而const_ptr是个指向（非常量）int的常量指针：

```
const int * ptr_to_const;
int * const const_ptr;
```

参考：数组声明器 第4.5.3节；const类型限定符 第4.4.4节；函数声明器 第4.5.4节；指针类型 第5.3节；类型限定符 第4.4.3节。

4.5.3 数组声明器

数组声明器用于声明数组类型的变量：

数组声明器：

直接声明器 [常量表达式$_{opt}$] （直到C99才可用）
直接声明器 [数组限定符类型$_{opt}$ 数组长度表达式$_{opt}$] （C99）
直接声明器 [数组限定符类型$_{opt}$ *] （C99）

常量表达式：

条件表达式

数组限定符列表：

数组限定符
数组限定符列表 数据限定符

数据限定符：

```
static
restrict
const
volatile
```

数组长度表达式：

赋值表达式
*

如果D是个封装了标识符id的任何声明器，并且“S D;”这个声明表示id的类型是“…S”，则下面这个声明：

```
S (D)[ e ] ;
```

表示id的类型是“S的…数组”（根据构建声明器的优先级规则，括号常常可以省略）。类型S不是不完整类型或函数类型。

在最常见的情况下，整数常量表达式e出现在方括号中，它指定了数组的元素数量。这个数必须是个大于0的整数。C的数组总是以0为基数。也就是说，int A[3]这个声明定义了数组元素A[0]、A[1]和A[2]。更高维数的数组被声明为“数组的数组”（参见第5.4.2节）。

例子

在下面这几个声明中，id是x，S是int，“…”分别是“”、“指向…的指针”和“…的数组”。

声明	x的类型
int (x) [5];	整型数组
int (*x) [5];	指向整型数组的指针

（续）

声明	x的类型
int (x[5]) [5];	整型数组的数组
int x[5][5];	整型数组的数组（与上一个同）

数组声明器的方括号内部并不一定需要出现整型表达式，还有3种可能的变型：不完整的数组类型，可变长度的数组以及在数组声明器中使用数组限定符（类型限定符和static）。

不完整的数组类型 如果方括号为空，则声明器描述了一种不完整的数组类型。不完整类型的对象是无法被创建的，因为它们的长度未知。但是，可以声明指向不完整类型的指针。下面是一些可以省略数组长度的情况：

1. 数组被声明为函数的形式参数。由于数组的形参将被转换为指针，所以不需要数组的长度。如果这个数组具有多维，只有最左边那维可以被省略。例如：

```
int f(int ary[]); /* array of unspecified length */
```

2. 声明器附有一个初始化值，可以根据这个初始化值推断数组的长度。在处理了这个初始化值之后，这个数组就不再是不完整类型。例如：

```
char prompt[] = "Yes or No?";
```

3. 声明并非作为定义出现，而是引用一个在其他地方声明的对象。在定义之后，这个对象就不再是不完整。对于多维数组，只有最左边那维可以省略长度。可以创建一个指向不完整类型的指针。例如：

```
extern int matrix[][10]; /* incomplete type */

...
static int matrix[5][10]; /* no longer incomplete */
```

4. 在C99中，结构的最后一个成员可以是灵活数组成员，它在声明时未指定长度。

任何n维数组的声明必须包含最后n−1维的各维长度，以便确定访问算法。

可变长度数组 在C99中，如果数组声明器的方括号中的数组长度表达式是个*或者是个并非常量的表达式，则这个声明器描述了一个可变长度的数组。*只能出现在函数原型的数组形参声明中，它并不是函数定义的一部分。可变长度数组并不是不完整类型。关于可变长度数组以及它们在函数原型中的用法，请参见第5.4.5节。

数组限定符 在C99中，允许在数组声明器的方括号中使用数组限定符列表，但只允许用于声明数组类型的函数形参。这个问题将在第9.3节讨论。

参考：数组类型 第5.4节；赋值表达式 第7.9节；条件表达式 第7.8节；常量表达式 第7.11节；灵活数组成员 第5.6.8节；形式参数 第9.3节；初始化值 第4.6节；引用和定义声明 第4.8节；类型限定符 第4.4.3节；可变长度数组 第5.4.5节。

4.5.4 函数声明器

函数声明器用于声明或定义函数，并用于声明那些把函数指针作为成员的类型：

函数声明器：

直接声明器（参数类型列表） (C89)

直接声明器（标识符列表$_{opt}$）

参数类型列表：

形参列表
形参列表,…

形参列表：

形参声明
形参列表,形参声明

形参声明：

声明指定符 声明器
声明指定符 抽象声明器$_{opt}$

标识符列表：

标识符
形参列表,标识符

如果D是个封装了标识符id的任何声明器，并且“S D;”这个声明表示id的类型是“…S”，则下面这个声明：

```
S (D) (P);
```

表示id的类型是“返回S并具有参数P的函数”。在大多数情况下，根据构建声明器的优先级规则，D周围的括号可以被省略（第4.5.5节）。声明器语法中如果存在参数类型列表，它表示这个声明器是个标准C的函数原型。如果没有参数类型列表，这个声明器就采用了传统形式，它可以同时被传统C编译器和标准C编译器所接受。

例子

下面显示了一些函数声明器的例子：

声 明	x的类型
int x();	未指定形参并返回一个整数的函数
int x(double, float);	接受一个double和一个float形参，并返回一个整数的函数（原型）
int x(double d, float f);	与前一个声明器相同
int (*x)();	是个函数指针，指向未指定形参并返回一个整数的函数
int (*x[])(int, …);	是个函数指针数组，每个函数指针指向接受以一个整数形参开始的可变数量的形参并返回一个整数的函数（原型）
int (* const x) (void)	是个常量函数指针，指向不接受参数并返回一个整数值的函数

函数声明器具有一些限制，取决于它们是出现函数定义中还是作为一个对象或函数类型声明的一部分。表4-5显示了函数声明器可能出现的形式，提示它是传统C形式还是标准C的原型形式、是否可以出现在函数定义或函数类型声明中，以及它们所指定的形参信息。在这张表中，T_x x这个记法所表示的语法是“声明指定符 声明器”（即一个包含形参名x的形参类型声明）。T_x表示

声明指定符 抽象声明器$_{opt}$

也就是省略了形参名称的形参类型声明。

表4-5 函数声明器

语 法	形式	出现于	所指定的形参
f()	传统	定义	无形参
f()	传统	类型声明	任何数量的形参
f(x, y, …, z)	传统	定义	固定[a]
f(void)	原型	均可	无形参
$f(T_x, T_y, \cdots, T_z)$	原型	类型声明	固定
$f(T_x, T_y, \cdots, T_z, \cdots)$	原型	类型声明	固定，加上额外的形参[b]
$f(T_x, T_y, \cdots, T_z z)$	原型	均可	固定
$f(T_x, T_y, \cdots, T_z z, \cdots)$	原型	均可	固定，加上额外的形参[b]

a 在标准C之前，可以具有额外的、未指定的形参。

b 额外形参的数量和类型并未指定。

第9章将更详细地讨论函数的声明和使用。可变长度的形参列表可以通过stdarg.h或varargs.h头文件中的工具来访问。

参考：抽象声明器 第5.12节；数组声明器 第4.5.3节。定义和引用声明 第4.8节；函数类型和声明 第5.8节；函数定义 第9.1节；指针声明器 第4.5节；stdarg.h和varargs.h 第11.4节。

4.5.5 声明器的组合

声明器可以进行组合，以构成更复杂的类型，例如“包含5个元素的数组，数组元素的类型是指向返回int的函数的指针”，下面这个声明中的ary就是这种类型：

```
int (*ary[5])();
```

声明器的唯一限制是最终的类型在C中必须是合法的。在C中不合法的类型包括：

1. 除了“返回void的函数”或“指向void的指针”（在标准C中）之外包含void的任何类型。
2. “函数的数组”数组可以包含指向函数的指针，但不能包含函数本身。
3. “返回数组的函数”函数可以返回指向数组的指针，但不能返回数组本身。
4. “返回函数的函数”函数可以返回指向其他函数的指针，但不能返回函数本身。

在组合声明器时，声明器表达式的优先级是非常重要的。函数和数组声明器的优先级高于指针声明器，因此“*x()”等于“*(x())”（“返回指向…的指针的函数”），而不是等于“(*x)()”（“指向返回…的函数的指针”）。可以使用括号对声明器进行正确的分组。早期的C编译器要求嵌套声明器的深度至少允许为6，标准C编译器允许嵌套声明器的最大深度至少为12。

尽管声明器可以是任意复杂，但作为一种良好的编程风格，把复杂声明器分解为几个更简单的声明器是更好的做法。

例子

声明	x的类型
int x();	返回一个整数的函数
int (*x) ();	指向返回一个整数的函数的指针
void (*x) ();	指向不返回任何值的函数的指针
void *x();	返回“指向void的指针”的函数

例子

不应该采用下面这种声明器组合：

```
int *(*(*(*x)())[10])();
```

而是把它们改写为：

```
typedef int *(*print_function_ptr)();
typedef print_function_ptr (*digit_routines)[10];
digit_routines (*x)();
```

变量x是个指向函数的指针，它所指向的函数返回一个指针，后者指向一个包含10个元素的数组，元素的类型是指向返回指向整数的指针的函数的指针。显然，这个声明令人茫然不知所措。

例子

声明器语法背后的准则是它们模仿了封装标识符的用法。为了说明声明和使用的对称性，如果看到下面这个声明：

```
int *(*x)[4];
```

则下面这个表达式的类型是int。

```
*(*x)[i]
```

参考：数组类型 第5.4节；函数类型 第5.8节；指向void的指针 第5.3.2节；指针类型 第5.3节；void类型限定符 第5.9节。

4.6 初始化值

在声明变量时，可以附有初始化值，后者指定了这个变量在它的生命期之初应该具有的值。初始化值的完整语法如下：

初始化值：

赋值表达式
{ 初始化值列表 ,$_{opt}$ }

初始化值列表：

初始化值
初始化值列表,初始化值
指定 初始化值 (C99)
初始化值列表,指定 初始化值 (C99)

指定：

指定器列表 =

指定器列表：

指定器
指定器列表 指定器

指定器：

[常量表达式]
. 标识符

花括号内可选的缀尾逗号并不会影响初始化值的含义。

C99允许指定初始化值（第4.6.9节）。在指定初始化值时，程序员可以指定聚合类型中需要进行初始化的特定成员。

一个特定声明所允许的初始化值取决于被初始化的对象的类型以及这个被声明的对象是否具有静态或自动存储类别。表4-6列出了这些选项，接下来几节将描述更多的细节。在外部对象的声明中，只有当它们是定义性声明（参见第4.8节）时，才应该具有初始化值。

初始化值的形式（由一对花括号所包围的初始化值列表）应该与被声明的变量的结构匹配。指定了标量变量的初始化值的语言定义可以加上花括号，尽管这种花括号在逻辑上是不需要的。我们推荐保留花括号用于聚合类型的初始化。有一些特殊的规则可以用于简化聚合类型的初始化值。

表4-6 初始化值的形式

存储类别	类型	初始化值表达式	默认初始化值
静态	标量	常量	0、0.0、false或null指针
静态	数组[a]或结构	花括号包围的常量（C99还允许非常量表达式）	每个成员的依次默认值
静态	联合[b]	常量（C99还允许非常量表达式）	第1个成员的默认值
自动	标量	任何值	无
自动	数组[a,b]	花括号包围的常量	无
自动	结构[b]	花括号包围的常量，或一个简单的同种结构类型的非常量表达式	无
自动	联合[b]	常量，或一个简单的同种联合类型的非常量表达式	无

a 数组可以具有未知的长度，由初始化值决定数组的长度。可变长度的数组无法被初始化。

b 标准C。旧式的编译器可能不允许对这类对象进行初始化。

历史说明：在C最初所使用的初始化值语法中，=操作符被省略。为了保持兼容，当前有些编译器仍然接受这种语法。当这类编译器的用户在声明中无意省略了一个逗号或分号时（例如“int a b;”），可能会收到一条晦涩的错误信息，表示非法的初始化值。标准C并不支持这种过时的风格。

下面各节解释了每种类型的变量的特殊需求。

参考：自动和静态生命期 第4.2节；声明 第4.1节；外部对象 第4.8节；静态存储类别 第4.3节。

4.6.1 整数

整数变量的初始化值的形式是：

声明器 = 表达式

初始化表达式必须是一种能够通过简单的赋值向被初始化的变量进行赋值的类型。在这个过程中，将会执行寻常赋值转换。如果变量为静态变量或外部变量，这个表达式必须是个常量。如果变量为自动变量或寄存器变量，就允许使用任何表达式。静态整数的默认初始化值为0。

例子

在下面的代码片段中，Count被初始化为一个常量表达式，但ch被初始化为一个函数调用的结果。

```
#include <stdio.h>
static int Count = 4*200;

int main(void)
{
    int ch = getchar();
    …
}
```

参考：常量表达式 第7.11节；整数类型 第5.1节；静态和自动范围 第4.2节；寻常赋值转换 第6.3.2节。

4.6.2 浮点数

浮点型变量的初始化值的形式是：

声明器 = 表达式

初始化表达式必须是一种能够通过简单赋值对被初始化的变量进行赋值的类型。在这个过程中，将会执行寻常赋值转换。如果变量为静态变量或外部变量，这个表达式必须是个常量。如果变量是自动变量或寄存器变量，就允许使用任何表达式。

例子

```
static void process_data(double K)
{
    static double epsilon = 1.0e-6;
    auto float fudge_factor = K*epsilon;
    …
}
```

标准C明确允许在初始化值中使用浮点常量表达式。有些旧式的C编译器可能不允许使用复杂的浮点常量表达式。

静态浮点变量的默认初始化值是0.0。这个值在有些目标计算机上可能并不是用所有的位都为0的对象表示。标准C编译器必须把变量初始化为0.0的正确表示形式，但大多数旧式的C编译器总是把静态变量初始化为所有的位均为0。

参考：算术类型 第5章；常量表达式 第7.11节；浮点常量 第2.7.2节；浮点类型 第5.2节；静态和自动范围 第4.2节；单目负号操作符 第7.5.3节；寻常赋值转换 第6.3.2节。

4.6.3 指针

指针变量的初始化值的形式是：

声明器 = 表达式

初始化表达式必须是一种能够通过简单赋值向被初始化的变量进行赋值的类型。在这个过程中，将会执行寻常赋值转换。如果变量为自动变量，就允许使用任何适当类型的表达式。

如果变量是静态变量或外部变量，表达式必须是常量。可以通过下面这些方式形成常量表达式，作为指针类型PT（指向T的指针）的初始化值。

1. 值为0的整数常量表达式，或者转换为void *的0值。这些值表示null指针，通常用标准函数库名称NULL表示。

```
#define NULL ((void *)0)
double *dp = NULL;
```

2. 类型为“返回T的函数”的静态或外部函数的名称被转换为一个“指向返回T的函数的指针”类型的常量。

```
extern int f();
static int (*fp)() = f;
```

3. “T数组”类型的静态或外部数组的名称被转换为一个“指向T的指针”类型的常量。

```
char ary[100];
char *cp = ary;
```

4. &操作符作用于一个T类型的外部或静态数组，产生一个“指向T的指针”类型的常量。

```
static short s; auto short *sp = &s;
```

5. &操作符作用于一个“T数组”类型的使用常量下标表达式的外部或静态数组，产生一个“指向T的指针”类型的常量。

```
float PowersOfPi[10];
float *PiSquared = &PowersOfPi[2];
```

6. 整数常量转换为指针类型，产生一个指针类型的常量，尽管这种做法是不可移植的。

```
long *PSW = (long *) 0xFFFFFFF0;
```

并不是所有的编译器都接受常量表达式的类型转换，但标准C允许这种做法。

7. 当一个字符串常量作为一个指针类型的变量的初始化值出现时，将产生一个“指向char的指针”类型的常量。

```
char *greeting = "Type <cr> to begin ";
```

8. 第3种情况到第7种情况的任何表达式与一个整数表达式的和或差。

```
static short s;
auto short *sp = &s + 3, *msp = &s - 3;
```

一般而言，指针类型的初始化值的结果必须是一个转换为指针类型的整数，或者是一个地址加上或减去一个整数常量的结果。这个限制反映了大多数链接器的功能。

静态指针的默认初始化值是null指针。在null指针不是用全0的位模式的对象表示的情况下，标准C规定必须使用正确的null指针值。大多数旧式的C编译器简单地把静态存储初始化为全0的位。

参考：取地址操作符& 第7.5.6节；数组类型 第5.4节；涉及指针的转换 第6.2.7节；函数类型 第5.8节；整数常量 第2.7节；指针声明器 第4.5节；指针类型 第5.3节；字符串常量 第2.7节；寻常赋值转换 第6.3.2节。

4.6.4 数组

如果I_j是个可以作为T类型对象的初始化值的表达式，则

$$\{ I_0 , I_1 , \ldots, I_{n-1} \}$$

可以作为“包含n个元素的T类型的数组”的初始化值。C99允许I_j是个非常量表达式，但以前版本的C要求它是常量表达式。初始化值I_j用于初始化数组的第j个元素（从0开始计数）。多维数组

采用相同的模式，逐行列出初始化值（在C中，最后一个下标的变化速度最快）。

例子

一维数组的初始化是通过列出它的元素的初始化值进行初始化的：

```
int ary[4] = { 0, 1, 2, 3 };
```

多维数组根据每个子数组进行初始化：

```
int ary[4][2][3] =
          { { { 0, 1, 2}, { 3, 4, 5} },
            { { 6, 7, 8}, { 9, 10, 11} },
            { {12, 13, 14}, {15, 16, 17} },
            { {18, 19, 20}, {21, 22, 23} } };
```

结构数组（第4.6.6节）也可以采用类似的方法进行初始化：

```
struct {int a; float b;} a[3] = { {1, 2.5},
                                  {2, 3.9},
                                  {0, -4.0} };
```

静态和外部数组总是可以按照这种方式进行初始化。标准C允许对自动数组进行初始化，但这个特性并不是C的最初定义。数组初始化具有一些特殊的规则：

1. 初始化值的数量可以少于数组元素的数量。在这种情况下，剩余的元素被初始化为它们的默认初始值（静态数组所使用的方式）。如果初始化值的数量大于数组的元素数量，就会出现错误。

例子

下面这些声明

```
float ary[5] = { 1, 2, 3 };
int mat[3][3] = { {1, 2}, {3} };
```

与下面这些声明等价

```
int ary[5] = { 1.0, 2.0, 3.0, 0.0, 0.0 };
int mat[3][3] = { {1, 2, 0},
                  {3, 0, 0},
                  {0, 0, 0} };
```

2. 数组的边界可以不必指定（就像不完整类型一样）。在这种情况下，数组的边界根据初始化值的数量来决定。这个情况同时适用于静态数组和自动数组的初始化。

例子

下面这个声明

```
int squares[] = { 0, 1, 4, 9 };
```

与下面这个声明等价

```
int squares[4] = { 0, 1, 4, 9 };
```

3. 字符串常量可以用于对“char数组”类型的变量进行初始化。在这种情况下，数组的第1个元素被初始化为字符串的第1个字符，接下来以此类推。如果有足够的空间或者未指定数组的长度，字符串的终止null字符‘\0’也被存储在数组中。字符串也可以出现在一对花括号中。如果字符串的长度大大超过字符数组的指定长度，并不会导致错误，但容易令人混淆（这在C++中是个错误）。

元素类型与wchar_t兼容的数组可以按照相同的方式用一个宽字符串常量进行初始化。

例子

下面这些声明：

```
char x[5]  = "ABCDE";
char str[] = "ABCDE";
wchar_t q[5] = L"A";
```

与下面这些声明等价

```
char x[5]   = { 'A', 'B', 'C', 'D', 'E' }; /* No '\0'! */
char str[6] = { 'A', 'B', 'C', 'D', 'E', '\0' };
wchar_t q[5]   = { L'A', L'\0', L'\0', L'\0', L'\0' };
```

4. 字符串列表可以用于对字符指针数组进行初始化。

例子

```
char *astr[] = { "John", "Bill", "Susan", "Mary" };
```

5. 可变长度的数组无法进行初始化。

参考：数组类型 第5.4节；字符常量 第2.7节；字符类型 第5.13节；指针类型 第5.3节；字符串常量 第2.7节；可变长度数组 第5.4.5节；宽字符串 第2.7.4节。

4.6.5 枚举

枚举类型的变量的初始化值的形式是：

声明器 = 表达式

初始化表达式必须是一种能够通过简单的赋值向被初始化的变量进行赋值的类型。在这个过程中，将会执行寻常赋值转换。如果变量是静态变量或外部变量，这个表达式必须是常量。如果变量是自动变量或寄存器变量，就允许使用任何表达式。

例子

作为一种良好的编程风格，建议初始化表达式的类型应该与被初始化的变量属于相同的枚举类型。例如：

```
static enum E { a, b, c } x = a;
auto enum E y = x;
```

参考：类型转换表达式第7.5.1；常量表达式 第7.11节；枚举类型 第5.5节；寻常赋值表达式 第6.3.2节。

4.6.6 结构

如果结构类型T具有n个有名称的成员，它们的类型是T_j（其中j = 1, …, n），并且I_j是个允许对T_j类型的对象进行初始化的初始化值，则

$$\{ I_1 , I_2 , \ldots, I_n \}$$

可以作为类型T的初始化值。无名的位段成员不参与初始化过程。在C99中，初始化值I_j不必是常量，但它们在以前版本的C中必须是常量。

例子

```
struct S {int a; char b[5]; double c; };
struct S x = { 1, "abcd", 45.0 };
```

所有的C编译器都支持对结构类型的静态和外部变量进行初始化。标准C允许对结构类型的自动和寄存器变量进行初始化，并且可以使用任何一种初始化形式。首先是由一对花括号所包围的常量表达式，就像静态变量所使用的初始化值一样。其次，也可以使用下面这种形式的初始化

声明器 = 表达式

其中，“表达式”的类型与被初始化的变量相同。有些旧式的C编译器在对包含位段的结构进行初始化方面存在一些缺陷。

和数组的初始化一样，结构的初始化值具有一些特殊的缩略规则。具体地说，如果初始化值的数量少于结构的成员数量，剩余的成员将被初始化为它们的默认值。如果为结构变量提供了过多的初始化值，就会出现错误。

例子

假设有下面这个结构声明

```
struct S1 {int a;
           struct S2 {double b;
                      char c; } b;
           int c[4]; };
```

下面的初始化

```
struct S1 x = { 1, {4.5} };
```

相当于

```
struct S1 x = { 1, { 4.5, '\0' }, { 0, 0, 0, 0 } };
```

参考：位段 第5.6.5节；常量表达式 第7.11节；结构类型 第5.6节。

4.6.7 联合

标准C允许对联合变量进行初始化（传统C不允许这种做法）。静态、外部、自动和寄存器类型的联合变量的初始化值必须是个出现在花括号内的常量表达式，并且允许对联合的第1个成员类型的对象进行初始化。自动或寄存器类型的联合变量的初始化值还可以是该联合类型的任何表达式。在C99中，可以使用指示符对第1个成员之外的其他成员进行初始化。

例子

下面显示了对联合变量x和y进行初始化的两种初始化值形式：

```
enum Greek { alpha, beta, gamma };
union U {
          struct { enum Greek tag; int Size; } I;
          struct { enum Greek tag; float Size; } F;
        };
static union U x = {{ alpha, 42 }};
auto union U y = x;
```

剩余的C类型还有函数类型和void。由于不允许声明这些类型的变量，所以它们不存在初始化问题。

参考：指定初始化值 第4.6.9节；静态范围 第4.2节；联合类型 第5.7节。

4.6.8 省略花括号

在某些情况下，C允许省略初始化值中的一些花括号，虽然保留花括号通常可以使初始化值看上去更清晰。下面列出了基本规则。

1. 如果一种数组或结构类型的变量被初始化，则最外层的花括号不能省略。

2. 对于非最外层花括号，如果初始化值列表所包含的初始化值的数量与需要被初始化的对象的元素数量相同，就可以省略花括号。

例子

这些规则最常用的场合是在初始化多维数组时省略内层花括号：

```
int matrix[2][3] = { 1, 2, 3,   4, 5, 6 };
      /* same as: { {1, 2, 3}, {4, 5, 6} } */
```

许多C编译器会随意处理初始化值列表，允许使用过多或过少的花括号。我们建议使初始化值保持简单，并使用花括号使它们的结构显得更清晰。

4.6.9 指定成员的初始化值

C99允许在初始化值列表中指定需要初始化的聚合类型（结构、联合或数组）的成员。指定成员的初始化值和位置性（非指定）初始化值可以混合出现在同一个初始化值列表中。

在数组的初始化值列表中，指示符采用[e]的形式，常量表达式e以索引方式指定了一个元素。如果这个数组的长度未指定，则允许使用任何非负索引值，并且那个最高的被初始化的索引值决定了这个数组的最终长度。如果一个位置性初始化值紧随一个指定成员的初始化值之后，则这个位置性初始化值在这个指定元素之后的位置开始进行初始化。按照这种风格，初始化值列表中后面的值可能会改写前面的值。

例子

下面每个初始化后面都有一个注释，提示了所有元素的最终初始值。

```
int a1[5] = { [2]=100, [1]=3 };
   /* {0, 3, 100, 0, 0} */
int a2[5] = { [0]=10, [2]=-2, -1, -3 };
   /* {10, 0, -2, -1, -3} */
int a3[] = { 1, 2, 3, [2]=5, 6, 7};
   /* {1, 2, 5, 6, 7} ; a3 has length 5 */
```

在结构的初始化值列表中，成员指定符采用.c的形式，其中c是结构成员的名称。如果一个位置性初始化值出现在指定成员的初始化值的后面，则这个位置性初始化值将在这个指定成员之后进行初始化。按照这种风格，初始化值列表中后面的值可能会改写前面的值。

例子

下面每个初始化后面都有一个注释，说明了所有成员的最终初始化值。

```
struct S {int a; float b; char c[4]; };
struct S s1 = { .c = "abc" };
   /* {0, 0.0, "abc"} */
struct S s2 = { 13, 3.3, "xxx", .b=4.5 };
   /* {13, 4.5, "xxx"} */
```

```
struct S s3 = { .c = {'a','b','c','\0'} };
   /* {0, 0.0, "abc"} */
```

在联合的初始化值列表中，成员指定符采用.c的形式，其中c是联合成员的名称。这就允许对联合的任何成员进行初始化，而不仅仅是它的第1个成员。

例子

下面每个初始化后面都有一个注释，说明了所有成员的最终初始化值。

```
union U {int a; float b; char c[4]; };
union U u1 = { .c = "abc" };
   /* u1.c is "abc\0"; other components undefined */
union U u2 = { .a = 15 };
   /* u2.a is 15; other components undefined */
union U u3 = { .b = 3.14 };
   /* u3.b is 3.14; other components undefined  */
```

嵌套的聚合类型也可以根据对应的风格对指定的成员进行初始化。成员指定符可以进行连接，以便对深层嵌套的元素进行初始化。

例子

下面每个初始化后面都有一个注释，说明了所有成员的最终初始化值。

```
struct Point {int x; int y; int z; };
typedef struct Point PointVector[4];
PointVector pv1 = {
   [0].x = 1, [0].y = 2, [0].z = 3,
   [1] = {.x = 11, .y=12, .z=13},
   [3] = {.y=3} };
   /* {{1,2,3},{11,12,13},{0,0,0},{0,3,0}} */
typedef int Vector[3];
typedef int Matrix[3][3];
struct Trio {Vector v; Matrix m; };
struct Trio t = {
   .m={[0][0]=1, [1][1]=1, [2][2]=1},
   .v={[1]=42,43} };
   /* {{0,42,43},{{1,0,0},{0,1,0},{0,0,1}}} */
```

4.7 隐式声明

在C99之前，在函数调用中所使用的外部函数并不需要在此之前已经声明。当编译器看到标识符id后面是个左括号，并且id以前并没有被声明时，在最内层的作用域中就会隐式地加入一个声明，形式如下：

```
extern int id();
```

如果id以前并没有被声明为函数，C99编译器将会产生一条诊断信息，但是通过隐式声明，代码仍然能够继续执行。有些非标准的编译器可能会在顶层而不是在最内层作用域声明这个标识符。

例子

允许函数的隐式声明是一种不良的编程风格，可能会导致错误，尤其是可能导致不正确的返回类型。如果一个函数返回一个指针，例如malloc（第16.1节），它可能被隐式地声明为：

```
extern int malloc();
```

而不是下面这种正确的形式：

```
extern char *malloc();  /* returns (void *) in Standard C */
```

如果int和char*是用不同的形式表示的，对malloc的调用很可能会出错。假设int类型占据2个字节而指针类型占据4个字节，当编译器看到下面这些的声明时

```
int *p;
…
p = (int *) malloc(sizeof(int));
```

它将会产生代码，把它以为的malloc所返回的2个字节的值扩充为指针所需要的4个字节。其效果是malloc所返回的地址只有低位的一半被赋值为p。如果已经分配了大量的内存，导致malloc所返回的地址大于0xFFFF时，程序就会出现错误。

4.8 外部名称

和外部名称有关的一个重要问题就是要保证同一个外部名称的声明在几个文件中的一致性。例如，如果同一个外部名称的两个声明指定了不同的初始化值，会发生什么情况呢？由于这个原因以及其他原因，在一组文件中区分一个外部名称的定义性声明是非常重要的。同一个名称的其他声明称为引用性声明，即它们引用了定义性声明。

C所存在的一个众所周知的缺陷是很难区分外部变量声明的定义和引用。一般而言，编译器使用4个模型之一来判断一个顶层声明是否为定义性声明。

4.8.1 初始化值模型

在顶层声明中，如果出现了初始化值，它就被认为是个定义性声明。其他的声明则是引用性声明。在C程序的所有文件中，同一个变量的定义性声明只能出现1次。这是标准C所采用的模型。另外，标准C还采用了下一节所讨论的另外一个规则。

4.8.2 省略的存储类别模型

在这种方案中，存储类别extern必须显式地出现在所有的引用性声明中，并且每个外部变量的唯一定义性声明必须省略这个存储类型指定符。定义性声明可以包含初始化值，但这不是强制要求。在一个声明中同时出现初始化值和存储类别extern是错误的。

在标准C中，没有存储类别或初始化值的顶层声明称为假设定义（tentative definition）。也就是说，它被看成是一个引用性声明。但是，如果文件中没有出现该变量的其他声明，则这个假设定义被认为是真正的定义。

在C++中，如果声明中存在初始化值，extern就会被忽略。

4.8.3 公共模型

这个方案称为“公共模型”，因为它和Fortran编程语言所采用的把一个COMMON代码块的所有引用合并到一个单独的定义声明相似。外部变量的定义性声明和引用性声明都具有存储类别extern，不管是显式还是默认。在链接在一起组成程序的所有目标文件中的每个外部对象的声明中，只有1个声明可以拥有初始化值。在链接时，同一个标识符的所有外部声明（在所有的C

目标文件中）进行组合，并确定唯一的定义性声明，而不必与任何特定的文件相关联。如果任何一个声明指定了初始化值，则这个初始化值就用于初始化这个数据对象（如果存在几个这样的声明，其结果是未定义的）。

这个解决方案对于程序员而言是最轻松的，在系统开发中最为实用。

4.8.4 混合公共模型

这个模型是“省略存储类别”模型和“公共”模型之间的一个交叉。UNIX的许多版本使用了这个模型。

1. 如果extern被省略并且存在初始化值，就确定它是这个标识符的定义性声明。如果组成程序的所有文件中存在2个或更多个这样的定义，就会在链接时或者在此之前就产生错误。

2. 如果extern被省略并且不存在初始化值，就触发一个Fortran的COMMOM风格的定义。同一个标识符的这类定义可以同时存在多个。

3. 如果存在extern，这个声明就被认为是对另一个定义性声明的引用。这种声明不能拥有初始化值。

如果没有为外部变量提供显式的初始化值，这个变量被初始化为整数常量0。

4.8.5 总结和建议

表4-7显示了一个顶层声明根据各种外部引用模型所作的解释。为了与大多数编译器保持兼容，我们建议遵循下面这些规则：

1. 每个外部变量具有一个单独的定义点（在源文件中）。在定义性声明中，省略extern存储类别，并提供一个显式的初始化值：

```
int errcnt = 0;
```

2. 在其他引用外部变量的每个源文件或头文件中，使用存储类别extern，并且不要包含初始化值：

```
extern int errcnt;
```

表4-7 顶层声明的解释

顶层声明	模型				
	初始化值	省略存储类型（以及C++）	公共	混合公共	标准C
int x;	引用	定义	定义或引用	定义或引用	引用[a]
int x = 0;	定义	定义	定义	定义	定义
extern int x;	引用	引用	定义或引用	引用	引用
extern int x = 0;	定义	（非法）	定义	（非法）	定义

a 如果文件中没有出现后续的定义，它可以成为定义声明。

另外，虽然与定义和引用的区分无关，但一个外部名称在组成程序的所有文件中应该被声明为相同的类型。C编译器无法验证不同文件中的声明是否具有相同的类型，如果出现了这种不一致，就会在运行时出现错误。UNIX系统的C编译器所提供的lint程序可以在多个文件中检查声明是否一致，有些商业的UNIX和Windows平台上的产品也提供了这个功能。

4.8.6 未引用的外部声明

尽管C语言并没有要求，但是习惯的做法是忽略从来没有被引用的外部变量或函数的声明。例如，如果“extern double fft();”出现在一个程序中，但fft函数从来没有被使用，则不会有对fft名称的外部引用传递给链接器。因此，fft函数将不会被加载到程序中，可以节省空间。

4.9 C++兼容性

4.9.1 作用域

在C++中，struct和union定义都是作用域。也就是说，出现在这些定义中的类型声明在外部是不可见的。标准C也采取了这种方式（第5.6.3节）。为了保持兼容，可以简单把类型声明移到结构的外面（有些C++编译器可能允许这种过时的做法，因为它不会产生歧义）。

例子

在下面的代码中，结构t是在结构s中定义的，但它是在这个结构的外面被引用的。这在C++中是非法的。

```
struct s {
   struct t {int a; int b;} f1; /* define t here */
} x1;
struct t x2; /* use t here; OK in C, not in C++ */
```

参考：作用域 第4.2.1节；结构的成员 第5.6.3节。

4.9.2 标签和typedef名称

结构和联合标签不应该作为typedef名称使用，除非它们表示和原来相同的标签类型。在C++中，标签名除了作为标签外，还被隐式地声明为typedef名称（但是，它们可能被同一个作用域中同一个名称的后续变量或函数声明所隐藏）。这可能会导致编译器产生诊断信息。在较为罕见的情况下，甚至可能出现不同的行为。

例子

下面是一些在C或C++中产生诊断信息的一些例子。

```
typedef struct n1 {…} n1;          /* OK in both C and C++ */
struct n2 {…}; typedef double n2; /* OK in C, not in C++ */
struct n3 {…}; n3 x; /* OK in C++, not in C */
```

但是，标签名称可以作为变量或函数名使用，而不会产生混淆。下面的声明序列在C和C++中都是合法的，尽管更好的做法是避免可能产生的混淆：

```
struct n4 {…};
int n4;
struct n4 x;
```

在C++中，内层作用域中struct标签的声明可能隐藏外层作用域的变量声明。这可能导致C程序的含义在没有警告的情况下就被修改。在下面的代码中，表达式sizeof(ary)表示C中数组的长度，但它在C++中表示struct类型的长度。

```
int ary[10];
```

```
…
void f(int x)
{
    struct ary { … };/* In C++, this hides previous ary */
    …
    x = sizeof(ary);  /* Different meanings in C and C++! */
}
```

关于C++中typedef重定义的兼容性，请参见第5.13.2节。

参考：名字空间 第4.2.4节；重定义typedef名称 第5.10.2节。

4.9.3 用于类型的存储类别指定符

不要在类型声明中使用存储类别指定符。它们在传统C中将被忽略，但在C++和标准C中被认为是非法的。

例子

```
static struct s {int a; int b;}; /* invalid */
```

参考：类型的存储类别 第4.4.2节。

4.9.4 const类型限定符

如果一个顶层声明具有类型限制符const，但没有显式的存储类别，它在C++中被认为是static，但在C中被认为是extern。为了保持兼容，应该检查顶层const声明，并提供显式的存储类别。

在C++中，字符串常量隐式地为const，但在C中并不如此。

例子

下面的声明在C和C++中具有不同的含义：

```
const int c1 = 10;
```

但是，下面这些声明在C和C++中具有相同的含义：

```
static const int c2 = 11;
extern const int c3 = 12;
```

在C++中，所有的const声明（除了那些引用外部定义的常量之外）都必须有一个初始化值。

参考：const类型限定符 第4.4.4节。

4.9.5 初始化值

在C++中，当一个字符串用于初始化一个固定长度的字符数组时（或用于初始化wchar_t数组的宽字符串常量），数组必须有足够的空间来容纳整个字符串，包含结尾的null字符。

例子

```
char str[5] = "abcde"; /* valid in C, not in C++ */
char str[6] = "abcde"; /* valid in both C and C++ */
```

4.9.6 隐式声明

C++和C99都不允许函数的隐式声明（第4.7节）。所有的函数都必须在使用之前进行声明。

参考：隐式声明 第4.7节。

4.9.7 定义和引用声明

在C++中，不存在顶层变量的假设定义。在C中被认为是假设定义的声明在C++中将被当做真正的定义。也就是说，下面的声明序列在标准C中是合法的，但在C++中将导致重复定义错误。

```
int i;
…
int i;
```

例子

这个规则也适用于静态变量，这意味着无法创建相互引用的、静态初始化的变量。

```
struct cell {int val; struct cell *next;};
static struct cell a; /* tentative declaration */
static struct cell b = {0, &a};
static struct cell a = {1, &b};
```

这对于全局变量而言不是问题：第1个static可以被替换为extern，第2个和第3个static可以被删除（C++也允许声明互相引用、静态初始化的变量，但它与C不兼容）。

参考：结构类型引用 第5.6.1节；假设声明 第4.8.2节。

4.9.8 函数链接

在C++中调用一个C函数时，这个函数必须声明为具有“C”链接。第10章将更详细地讨论这个细节。

例子

在一个C++程序中，如果想调用一个由C编译器所编译的函数f，必须像下面这样编写（C++）声明：

```
/* This is a C++ program. */
extern "C" int f(void);/* f is a C, not C++, function */
```

4.9.9 无参函数

在C++的函数声明中，如果形参列表为空，表示这个函数不接受任何参数，但在C中被理解为接受未指定数量的参数。也就是说，C++的声明int f()与C的声明int f(void)相同。

4.10 练 习

1. 下面显示了静态函数P的定义。如果P在此之前从来没有被调用过，P(6)的值将是什么？如果P(6)第2次被调用，它的值又是什么？

```
static int P(int x)
{
    int i = 0;
    i = i+1;
    return i*x;
}
```

2. 下面这个程序片段显示了一个代码块，它包含了名称f的各种声明。这些声明是否存在冲突？如果存在，去掉冲突的声明，使程序合法。在程序中尽可能多地保留f的不同声明。

```
{
    extern double f();
    int f;
    typedef int f;
    struct f {int f,g;};
    union f {int x,y;};
    enum {f,b,s};
    f: ...
}
```

3. 下面的程序片段声明了3个名称为i的变量，它们的类型分别是int、long和float。每个变量分别是在哪一行被声明和使用的？

```
 1 int i;
 2 void f(i)
 3     long i;
 4 {
 5     long l = i;
 6     {
 7         float i;
 8         i = 3.4;
 9     }
10     l = i+2;
11 }
12 int *p = &i;
```

4. 编写C声明，表达下面的含义。在函数声明中使用原型。
 (a) P是个外部函数，不接受任何参数，并且没有返回值。
 (b) i是个局部整型变量，它将被大量使用，所以应该进行优化以提高访问速度。
 (c) LT是“指向字符的指针”的同义词。
 (d) Q是个外部函数，它接受2个参数，但没有返回值。第1个参数i是个整数，第2个参数cp是个字符串。这个字符串将不能被修改。
 (e) R是个外部函数，它的唯一参数是个函数指针。它所指向的函数接受1个32位的整型参数i，并返回一个指向double类型的指针。R返回一个整数值。假设long类型的宽度为32位。
 (f) STR是个静态、未初始化的字符串，它应该可以被修改，最多可以保存10个字符，不包括结尾的null字符。
 (g) STR2是个字符串，它被初始化为INIT_STR2宏的值所表示的字符串常量。在初始化之后，这个字符串将不能再被修改。
 (h) IP是个指向整数的指针，被初始化为变量i的地址。
5. 矩阵m被声明为int m[3][3]；第1个下标指定了行号，第2个下标指定了列号。为m编写一个初始化值列表f，把第1行初始化为1，把第2行初始化为2，把第3行初始化为3。
6. 假设有下面的定义：

```
const int * cip;
int * const cpi;
int i;
int * ip;
```

下面哪些赋值是允许的？

```
(a) cip = ip;
(b) cpi = ip;
(c) *cip = i;
(d) *cpi = i;
```

7. 使用C99的指定成员的初始化值形式，为一个包含3 × 3的int元素的矩阵identity编写声明和初始化值。这个初始化值应该把1赋值给元素identity[1][1]、identity[2][2]和identity[3][3]，并把0赋值给所有其他元素。
8. 为两个结构标签分别为left和right的结构编写声明。left结构应该依次包含一个称为data的double字段和一个指向right结构的指针link。right结构应该依次包含一个称为data的int字段和一个指向left结构的指针link。
9. 假设读者刚刚购买了一个C99编译器，并用它编译原来的软件。这个软件用旧式的C89编译器进行编译时并不会产生错误，但这个C99编译器报告了一些问题。对于下面所报告的每个错误，解释它们的原因：

 (a) C99编译器拒绝了一个函数调用，报告这个函数并没有被定义。

 (b) C99编译器拒绝了局部变量声明register i;。
10. 在读者的C程序中，假设fm被定义为一个类似函数的宏，om被定义为一个类似对象的宏（第3.3节）。如果这个函数还包含了下面的局部变量声明：

```
int fm;
int om;
```

这两个声明以及这些宏之间会不会出现冲突？讨论当这个程序被编译时可能出现的情况。

第5章 类 型



类型就是一组值以及在这些值上的一组操作。例如，整数类型的值由某个指定范围内的整数所组成，对这些值的操作包括加法、减法和不相等性测试等。浮点类型所包含的值与整数类型不同，并且具有一组不同的操作，例如浮点加法、减法和不相等性测试等。

当一个变量或表达式的值限制在“类型T”的范围之内时，我们就认为这个变量或表达式的类型为T。变量的类型是在声明变量时确定的；表达式的类型是由表达式操作符的定义决定的。C语言提供了大量的内建类型，包括几种整数类型、浮点数类型、指针、枚举、数组、结构、联合和函数。

按表5-1的方式对C的类型进行组织是很有用的。整数类型包括所有形式的整数、字符和枚举。算术类型包括整数类型和浮点类型。标量类型包含算术类型和指针类型。函数类型表示“返回……的函数”类型。聚合类型包括数组和结构。联合类型是用union指定符创建的。void类型没有值，并且没有任何操作。

表5-1 C的类型及其分类

<table>
<tr><th>C类型</th><th>类型分类</th><th></th><th></th></tr>
<tr><td>short, int, long, long long
（有符号和无符号）</td><td rowspan="4">整数类型</td><td rowspan="6">算术类型[a]</td><td rowspan="7">标量类型</td></tr>
<tr><td>char（有符号和无符号）</td></tr>
<tr><td>_Bool [b]</td></tr>
<tr><td>enum {...}</td></tr>
<tr><td>float, double,
long double</td><td rowspan="2">浮点类型</td></tr>
<tr><td>float _Complex,
double _Complex,
long double _Complex,
float _Imaginary,
double _Imaginary,
long double _Imaginary [b]</td></tr>
<tr><td>T *</td><td colspan="2">指针类型</td></tr>
<tr><td>T [...]</td><td colspan="2">数组类型</td><td rowspan="2">聚合类型</td></tr>
<tr><td>struct {...}</td><td colspan="2">结构类型</td></tr>
<tr><td>union {...}</td><td colspan="3">联合类型</td></tr>
<tr><td>T (...)</td><td colspan="3">函数类型</td></tr>
<tr><td>void</td><td colspan="3">void类型</td></tr>
</table>

a 除了复数之外的所有算术类型都属于实数类型。

b C99所新增。_Imaginary是可选的。

_Bool、_Complex和_Imaginary类型是C99新增的。布尔类型（_Bool）是一种无符号整数类型，而6种复数类型则是浮点类型。C99进一步把算术类型分成域（domain）：6种复数类型属于

复数域，所有其他算术类型位于实数域，属于实数类型。

本章将讨论C的所有类型。对于每种类型，我们说明如何声明这种类型的对象、这种类型的值的范围、这种类型的长度或表示形式方面的任何限制以及在这种类型的值上所定义的操作。

参考：数组类型 第5.4节；布尔类型 第5.1.5节；字符类型 第5.1.3节；复数类型 第5.2.1节；声明 第4.1节；枚举类型 第5.5节；浮点类型 第5.2节；函数类型 第5.8节；整数类型 第5.1节；指针类型 第5.3节；联合类型 第5.7节；void类型 第5.9节。

5.1 整数类型

C提供了比其他语言更多的整数类型和操作符。这些变型反映了在绝大多数计算机上所存在的不同字长和各种类型的算术操作符，使C程序和底层硬件之间具有紧密的对应。C的整数类型用于表示下面这些数据：

1. 有符号或无符号整数值，为它们提供了普通的算术和关系运算。
2. 位向量，对它们的操作包括“非”、“与”、“或”以及左移位和右移位操作。
3. 布尔值，0被认为是“假”，所有非零值被认为是“真”，其中整数1是最规范的“真”值。
4. 字符，它在计算机上是用整数编码表示的。

枚举类型也是与整数相似的类型，它们将在第5.5节讨论。

标准C要求编译器使用整数的二进制编码。这个要求说明了一个事实，就是许多低层的C操作在不存在二进制表达形式的计算机上无法以任何可移植的方式进行描述。

整数类型可以方便地分为4种：符号类型、无符号类型、布尔类型和字符类型。对于每种类型，都可以使用一组类型指定符声明这种类型的对象。

整数类型指定符：

```
有符号类型指定符
无符号类型指定符
字符类型指定符
布尔类型指定符                    (C99)
```

5.1.1 有符号整数类型

C向程序员提供了4种标准的有符号整数类型，分别用类型指定符short、int、long和long long表示（按照非降序排列）。signed char是第5种有符号整数类型，但它将在第5.1.3节讨论。C99引入了long long类型以及扩展的整数类型（第5.1.4节）。

每种类型具有几种等价的命名方式。在下面的语法中，分别显示了这4种类型的几个等价名称。

有符号类型指定符：

```
short或short int或signed short或signed short int
int或signed int或signed
long或long int或signed long或signed long int
long long或long long int或signed long long或signed long long int
```

关键字signed是C89新增的，它可以省略，以便与旧式的C编译器兼容。signed存在与否唯一可能影响程序含义的地方是，当它与char类型一起使用以及当它在结构的位段中使用时。在这种

情况下，有符号整数和“普通整数”（即在声明时未使用signed关键字的整数）之间可能存在区别。

标准C指定了大多数整数类型的最小精度。char类型至少必须达到8位，short类型的宽度至少为16位，long类型的宽度至少为32位，long long类型的宽度至少为64位（也就是说，C99要求64位的整数类型以及64位算术的完整集合）。整数类型的实际范围是在limits.h中记录的。

有符号整数类型可以表示的值的精度范围不仅取决于这种表示形式所使用的位数，而且取决于编码技术。到目前为止，整数的绝大多数常用的二进制编码技术称为补码，这是一种有符号的用n个位所表示的整数，它的取值范围从-2^{n-1}到$2^{n-1}-1$。它编码的方式如下：

1. 字的最高位（最左端）是符号位。如果符号位是1，这个数就是负数。否则，这个数就是正数。

2. 正数遵循下面的常规二进制序列：

$0 = 000...0000_2$
$1 = 000...0001_2$
$2 = 000...0010_2$
$3 = 000...0011_2$
$4 = 000...0100_2$
...

在一个n位的字中，去掉符号位，共有n−1位可以表示正整数。它可以表示0到$2^{n-1}-1$之间的整数。

3. 为了求一个整数的负数，对这个字的所有位进行反转，并把最后结果加1。因此，为了形成整数−1，首先从1（$000...0001_2$）开始，然后对所有的位求反（$111...1110_2$），最后加上1（$111...1111_2 = -1$）。

4. 最大的负数是$10...0000_2$或-2^{n-1}，它不存在对应的正数。对这个数求反所得到的是同一个数。

其他二进制整数编码技术还有反码记法，就是简单地对字的所有位求反。符号绝对值记法就是简单地对符号位求反。这些编码方案可以表示的值的范围从$-(2^{n-1}-1)$到$2^{n-1}-1$。它们可以表示的负值要少1个，0具有两种表示形式（正零和负零）。这三种编码方案用相同的方式表示正值，它们在标准C中都是可以接受的。

标准C要求在limits.h头文件中记录整数类型的范围。这个头文件还指定了遵循ISO的编译器所实现的每种整数类型可以使用的最大值。表5-2显示了limits.h头文件必须定义的一些符号。不同的编译器可以选择使用自己的值，但它们所选择的值的绝对大小不得小于表中所列出的值，并且必须具有相同的符号。因此，遵循ISO的编译器不能够只用8位来表示int类型。严格遵循标准的C程序不能依赖像−32768这样的值可以用short类型来表示（使用二进制整数反码表示形式的计算机支持用short类型表示这个值）。使用非ISO编译器的程序员可以创建一个limits.h头文件，定义这种编译器所使用的各种值。这个头文件所记录的范围并不一定和类型的实际大小相同，因为可能存在填充位（参见第 6.1.6节）。

C89修正案1添加了WCHAR_MAX和WCHAR_MIN，分别指定了wachr_t类型可以表示的最大值和最小值。但是，这些符号是在wchar.h头文件中定义的，而不是在limits.h中定义的。C99增加了一个新的头文件stdint.h，记录了整数类型的其他限制。

表5-2 limits.h所定义的值

名 称	最小值	含 义
CHAR_BIT	8	char类型的位数，以位为单位
SCHAR_MIN	$-(2^7-1)$；−127	signed char的最小值
SCHAR_MAX	2^7-1；127	signed char的最大值
UCHAR_MAX	2^8-1；255[a]	unsigned char的最大值
SHRT_MIN	$-(2^{15}-1)$；−32 767	short int的最小值
SHRT_MAX	$2^{15}-1$；32 767	short int的最大值
USHRT_MAX	$2^{16}-1$；65 535	unsigned short的最大值
INT_MIN	$-(2^{15}-1)$；−32 767	int的最小值
INT_MAX	$2^{15}-1$；32 767	int的最大值
UINT_MAN	$2^{16}-1$；65 535	unsigned int的最大值
LONG_MIN	$-(2^{31}-1)$；−2 147 483 647	long的最小值
LONG_MAX	$2^{31}-1$；2 147 483 647	long的最大值
ULONG_MAX	$2^{32}-1$；4 294 967 295	unsigned long的最大值
LLONG_MIN	$-(2^{63}-1)$；−9 223 372 036 854 775 807	long long的最小值
LLONG_MAX	$2^{63}-1$；+9 223 372 036 854 775 807	long long的最大值
ULLONG_MAX	$2^{64}-1$；18 446 744 073 709 551 615	unsigned long long的最大值
CHAR_MIN	SCHAR_MIN或0[b]	char的最小值
CHAR_MAX	SCHAR_MAX或UCHAR_MAX[c]	char的最大值
MB_LEN_MAX	1	在任何得到支持的本地化设置中一个多字节字符的最大字节数

a UCHAR_MAX必须是$2^{CHAR_BIT}-1$。

b 如果char类型在默认情况下是有符号的，则是SCHAR_MIN，否则为0。

c 如果char类型在默认情况下是有符号的，则是SCHAR_MAX，否则为UCHAR_MAX。

例子

下面是一些典型的有符号整数声明的例子：

```
short i, j;
long int l;
static signed int k;
```

为了使程序尽量能够移植，最好不要依赖int可以表示−32 767～32767范围之外的整数。如果这个范围无法满足要求，可以改用long类型。使用typedef根据每个特定程序的需要定义特殊的整数类型通常是种很好的风格。例如：

```
/* invdef.h Inventory definitions for the XXX computer. */
typedef short part_number;
typedef int   order_quantity;
typedef long  purchase_order;
```

C99的最佳解决方案是使用一种扩展整数类型名称，指定所需的精度。

```
/* invdef.h Inventory definitions for the XXX computer. */
#include <stdint.h>
typedef uint_least64_t part_number;      // at least 64 bits
typedef int_fast32_t   order_quantity;   // fast and 32 bits
typedef int32_t        purchase_order;   // exactly 32 bits
```

例子

在C中，所有的整数都可以用来表示布尔值。0表示“假”，所有非零值都表示“真”。布尔表达式如果是假，则它的求值结果是0。如果布尔表达式为真，则它的求值结果为1。例如，i = (a < b)，后面这个布尔表达式的值被赋值给整型变量i。如果a小于b，这个值是1。如果a并不小于b，这个值是0。类似，下面的语句：

```
if (i) statement1; /* Do this if i is nonzero */
else statement2;   /* Do this if i is zero */
```

如果i是非零值（真），$statement_1$就被执行。如果i等于0，$statement_2$就被执行。

C99引入了一种真正的布尔类型_Bool。它还引入了一个头文件stdbool.h，定义了更便于使用的类型名称bool以及布尔值true和false。这些名称与C++的布尔类型一致，但与C所使用的传统宏名FALSE和TRUE不同。

```
#include <stdbool.h>;
bool b;
…
b = (x < y) && (y < z);
if (b) …
```

不使用C99编译器的程序员也可以方便地定义bool、true和false。

参考：结构中的位段 第5.6.5节；_Bool类型 第5.1.4节；声明 第4.1节；扩展整数类型 第5.1.4节；整数常量 第2.7.1节；signed类型指定符 第5.1.1节；stdbool.h 第11.3节；stdint.h 第21章；类型转换 第6章；typedef 第5.10节。

5.1.2 无符号整数类型

对于每一种有符号整数类型，都存在一种对应的无符号类型，它占据相同数量的存储空间，但采用了不同的整数编码方式。为了声明无符号类型的整数，可以在对应的有符号类型指定符前面添加关键字unsigned为前缀（如果原来有signed关键字，则将它取代）。

无符号类型指定符：

```
unsigned short int_opt
unsigned int_opt
unsigned long int_opt
unsigned long long int_opt                    (C99)
```

int关键字是可选的，但推荐使用。在选择无符号整数类型时需要考虑的事项与有符号整数类型相同。C99引入了unsigned long long int和_Bool类型，后者也被认为是一种无符号类型，但在其他地方讨论。

所有的无符号类型都使用最直接的二进制记法，而不管有符号类型所使用的是补码、反码还是符号绝对值记法。符号位被看成是普通的数据位。因此，一个n位的字可以表示0到2^n-1的整数。大多数计算机很容易解释一个字的值，不管使用它所使用的是有符号记法还是无符号记法。例如，在使用补码时，位模式$11\ldots1111_2$（长度为n）可以表示−1（使用有符号记法）或2^n（使用无符号记法）。不论是有符号记法还是无符号记法，0到$2^{n-1}-1$范围内的整数表示方法是相同的。标准C编译器所采用的无符号类型的特定范围是在limits.h头文件中记录的。

一个整数是有符号还是无符号类型会影响它的运算。无符号整数的所有算术运算都是根据

同余模数2^n进行的。例如，一种无符号类型的最大值加1的结果保证是0。这就很好地定义了溢出的行为。

如果表达式中混合了有符号和无符号整数，将会强制使用无符号运算。第6.3.4节将讨论其中所执行的转换，第7章将讨论当参数为无符号整数时每个操作符的效果。

例子

这种转换可能会产生令人吃惊的结果。例如，由于无符号整数总是非负的，读者可能会以为下面这个测试的结果总是真的：

```
unsigned int u;
…
if (u > -1) …
```

但是，它的结果总是假的！有符号整数−1在进行比较之前总是会被转换为无符号类型，结果是最大的无符号整数，而u的值不可能大于这个整数。

在C的最初定义中，只提供了一种无符号类型unsigned。大多数非标准C编译器现在也提供了所有范围的无符号类型。

参考：_Bool类型 第5.1.5节；整数转换 第6.2.3节；常量 第2.7节；limits.h 第5.1.5节；有符号类型 第5.1.1节。

5.1.3 字符类型

C的字符类型是一种整数类型。也就是说，这种类型的值都是整数，可以在整型表达式中使用：

字符类型指定符：

```
char
signed char
unsigned char
```

字符类型有3种变体：有符号字符、无符号字符和普通字符。它们所占据的存储空间相同，但可能表示不同的值。有符号和无符号字符表示形式的用法与有符号和无符号整数表示形式相同。如果类型指定符中没有singed和unsigned，字符的类型就是普通的。signed关键字是标准C新增的，因此在不认识这个关键字的C编译器中，只有2种字符类型：无符号字符和普通字符。在C中，字符数组的概念就是“字符串”。

例子

下面是一些和字符有关的典型声明。

```
static char greeting[7];        /* a 7-character string */
char *prompt; /* a pointer to a character */
char padding_character = '\0'; /* a single character */
```

字符类型的表示形式取决于字符的本质以及目标计算机的字符串处理功能。字符类型具有一些特殊的性质，使它与普通的有符号和无符号类型有所不同。例如，普通的char类型可能是有符号的，也可能是无符号的，或者是两者的混合。由于效率的原因，C编译器可以自由地按照下面这两种方式之一来处理char类型：

1. char类型可以是与signed char相同的有符号整数类型。

2. char类型可以是与unsigned char相同的无符号整数类型。

在有些非标准的C编译器中，char类型是一种“伪无符号”整数类型。也就是说，它只能包含非负值，但是在执行寻常单目转换时，它总是被当成是有符号类型。

例子

如果需要真正的无符号字符类型，可以使用unsigned char类型。如果需要真正的有符号字符类型，可以使用signed char类型。如果char类型使用8位补码表示形式，并且有下面的声明

```
unsigned char uc = -1;
signed char sc = -1;
char c = -1;
int i = uc, j = sc, k = c;
```

则在所有的标准C编译器中，i的值必须是255，j的值必须是−1。但是，k的值到底是255还是−1则因编译器而异。如果C编译器并不认识关键字signed或者不允许unsigned char，结果就是有歧义的普通字符。

字符有无符号是非常重要的，因为标准I/O库函数一般返回文件中的字符，当它到达文件尾时会返回一个负值（标准头文件通过EOF宏指定了一个负值，通常是−1）。程序员应该把这些函数的返回值看成是int类型，因为char类型可能是无符号的。

例子

下面这个程序的目的是从标准输入流把字符复制到标准输出流，直到getchar返回文件尾提示符。前三个定义一般是在标准头文件stdio.h中定义的：

```
extern int getchar(void);
extern void putchar(int);
#define EOF (-1) /* Could be any negative value */
void copy_characters(void)
{
    char ch; /* Incorrect! */
    while ((ch = getchar()) != EOF)
        putchar(ch);
}
```

但是，当char为无符号类型时，这个函数就无法完成任务。为了理解这个效果，假设char类型用8位表示，int类型用16位表示，并且使用补码记法。当getchar返回−1时，ch = getchar()这个赋值将把255（−1的低8位）赋值给ch。这个循环的测试条件变成了255 != −1。如果char类型是无符号的，寻常转换将导致−1被转换为一个无符号整数，使测试条件成为255 != 65536，结果是“真”。因此，这个循环永远不会终止。把ch的声明修改为“int ch”就解决了这个问题。

例子

为了提高可读性，可以定义一种“伪字符”类型，用于这类情况。例如，下面重新编写的copy_characters使用了新类型character，它是用int类型所表示的字符：

```
typedef int character;
…
void copy_characters(void)
{
    character ch;
    while ((ch = getchar()) != EOF)
        putchar(ch);
}
```

字符的第2个含糊之处在于它的长度。在前面这个例子中，我们假定它们占据8位。这个假设几乎总是有效的，尽管仍不清楚它们的取值范围是从0～255还是从−127（或−128）到127。但是，有些计算机可能使用9位甚至32位来表示字符，程序员必须警惕。标准C要求编译器在limits.h头文件中记录字符类型的范围。

参考：位段 第5.6.5节；字符常量 第2.7.3节；字符集 第2.1节；EOF 第15.1节；getchar 第15.6节；整数类型 第5.1节；整数转换 第6.2.3节；limits.h 第5.1.1节。

5.1.4 扩展整数类型

在C99中，除了“标准”的整数类型之外，编译器还可以使用“扩展的”有符号整数类型。每种扩展有符号整数类型必须具有一个对应的无符号类型。为这些类型所选择的关键字必须以两个下划线开始，或者以一个下划线和一个大写字母开始（在标准C中，这类标识符被保留，可能由编译器使用）。这些扩展类型被认为是整数类型。所有适用于标准整数类型的说明也适用于这些扩展整数类型。扩展整数类型是在C99的头文件stdint.h和inttypes.h中定义的，第21章将描述这两个头文件。

标准整数转换也适用于扩展整数类型。这些规则是在第6章讨论转换级别时描述的。

参考：转换级别 第6.3.3节；有符号整数类型 第5.1.1节。

5.1.5 布尔类型

C99引入了无符号整数类型_Bool，它能够容纳的值只有0和1（分别表示“假”和“真”）。其他整数也可以在布尔上下文环境（例如条件表达式中的测试）中使用。但是，如果编译器遵循C99，使用_Bool可以使代码更为清晰。把任何标量值转换为_Bool类型时，所有非零值被转换为1，零值被转换为0。

stdbool.h头文件定义了bool宏，它与_Bool同义。这个文件还定义了false和true，它们的值分别是0和1。bool并不是关键字，这是为了保护旧式C程序，因为它们可能使用bool作为用户定义的类型名称。和_Bool有关的转换将在讨论其他整数转换和提升时描述。

参考：整数转换 第6.2.3节；整型提升 第6.3.3节；stdbool.h 第11.3节。

5.2 浮点类型

按照传统，C的浮点数具有两种长度，即单精度和双精度，或者称为float和double。标准C添加了long double类型，C99增加了3种复数浮点类型（第5.2.1节）。非复数浮点类型又称为实数浮点类型。

浮点类型指定符：

```
float
double
long double                                    (C99)
复数类型指定符                                  (C99)
```

旧式编译器允许使用类型指定符long float，它与double同义，但极少使用。标准C删除了这个类型指定符。

例子

下面是一些浮点类型对象的典型声明：

```
double d;
static double pi;
float coefficients[10];
long double epsilon;
```

使用float、double和long double好比是使用short、int和long。在标准C之前，编译器在执行任何浮点运算之前需要把所有float类型的值都转换为double（参见第6.3.4节），因此使用float类型的效率并不比使用double类型高。在标准C中，可以直接使用float类型执行浮点运算。在C99中，有一组完整的库函数支持float类型。

C并没有指定浮点类型的长度，甚至没有指定几种浮点类型的长度是否不同。程序员可以认为float类型可以表示的值是double类型可以表示的值的一个子集，而后者又是long double的一个子集。有些C编译器假设double可以精确地表示long类型的所有值。也就是说，把一个long类型的对象转换为double类型，然后再转换回long类型，所得到的结果和最初的long值相同。尽管在大多数情况下确实如此，但程序员不能依赖于此。

标准C要求在float.h头文件中记录实数浮点类型的特征。表5-3列出了必须定义的符号。以FLT开头的符号表示float类型，以DBL开头的符号表示double类型，以LDBL开头的符号表示long double类型。表5-3还显示了每个符号所允许的数量级，也就是浮点类型的范围和精度的最低要求。

表5-3 float.h所定义的值

名 称	最小值	含 义
FLT_RADIX[a]	2	基数b的值
FLT_ROUNDS[a]	无	四舍五入模式。-1：表示不确定；0：表示趋向0；1：取最接近的值；2：趋向正无穷；3：趋向负无穷[b]
FLT_EVAL_METHOD[c]	无	-1：不确定；0：只用于类型本身的范围和精度；1：float和double使用double，long double使用自身；2：long double用于所有求值
FLT_EPSILON	10^{-5}	最小值$x > 0.0$，使$1.0 + x > 1.0$；b^{1-p}；这里所显示的是所允许的最大值
DBL_EPSILON	10^{-9}	
LDBL_EPSILON	10^{-9}	
FLT_DIG	6	小数点后的数字位数，表示精度
DBL_DIG	10	
LDBL_DIG	10	
FLT_MANT_DIG DBL _MANT_DIG LDBL _MANT_DIG	无	p，有效数字中基数b的数字数量
DECIMAL_DIG[c]	10	表示所支持的最宽的浮点类型的值所需要的十进制数字的数量 等于$1 + p_{max}\log_{10}b$（如果b不是10的整数次方）
FLT_MIN	10^{-37}	最小的规格化正数
DBL_MIN	10^{-37}	
LDBL_MIN	10^{-37}	
FLT_MIN_EXP DBL_MIN_EXP	无	e_{min}，最小的负整数x，使b^{x-1}位于规格化浮点数的范围之内

（续）

名 称	最小值	含 义
LDBL_MIN_EXP		
FLT_MIN_10_EXP	−37	x的最小值，使10^x位于规格化浮点数的范围之内
DBL_MIN_10_EXP	−37	
LDBL_MIN_10_EXP	−37	
FLT_MAX	10^{+37}	可以表示的最大有限数
DBL_MAX	10^{+37}	
LDBL_MAX	10^{+37}	
FLT_MAX_EXP	无	e_{max}，整数x的最大值，使b^{x-1}是个可表示的有限浮点数
DBL_MAX_EXP		
LDBL_MAX_EXP		
FLT_MAX_10_EXP	37	x的最大值，使10^x位于有限浮点数可以表示的范围之内
DBL_MAX_10_EXP	37	
LDBL_MAX_10_EXP	37	

a FLT_RADIX和FLT_ROUND适用于所有3种浮点类型。

b 其他值是由编译器所定义的。

c C99所新增。

大多数算术和逻辑操作可以应用于浮点类型的操作数。这些操作包括算术和逻辑求反、加法、减法、乘法、除法、关系和相等性测试、逻辑AND和OR、赋值以及所有算术类型之间的转换。

采用符号绝对值表示形式并且没有“隐藏”位的实数浮点数x可以写成：

$$x = s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k}, e_{min} \leqslant e \leqslant e_{max}$$

其中：

s 符号（±1）

b 基数（一般是2、8、10或16）

e 是e_{min}和e_{max}之间的指数值

p 有效数字中基数b的数字位数

f_k 表示有效数字，$0 \leqslant f_k < b$

规格化浮点数满足$f_1 > 0$（x不等于0）。次规范数（subnormal number）就是$e = e_{min}$并且$f_1 = 0$的非零数。非规格数（unnormalized number）就是$e > e_{min}$并且$f_1 = 0$的非零数（次规格数由于太小，无法进行规格化。非规格数可以进行规格化，但由于某些原因而没有进行）。

浮点类型可以包括并非浮点数的特殊值：无限和NaN（不是数）。安静型NaN（quiet NaN）是在算术表达式中产生的，并不会导致异常。包含NaN的表达式的结果是NaN。表达式在遇到信号型NaN（signaling NaN）时会产生异常。无限和NaN可以带符号，并且NaN可以有许多变型。C99对标准库进行了扩展，允许输入和输出这些特殊的值，并且提供了一些库函数，用于创建和测试这些值（第17.13节和第17.14节）。

例子

许多微处理器所使用的一种常用的浮点数表示形式是由表示二进制浮点运算的IEEE标准

(ISO/IEEE标准754-1985) 指定的。根据这个标准，32位的单精度浮点数和64位的双精度数（调整为标准C记法约定）是：

$$x_{\text{float}} = s \times 2^e \times \sum_{k=1}^{24} f_k \times 2^{-k} \quad -125 \leqslant e \leqslant +128$$

$$x_{\text{double}} = s \times 2^e \times \sum_{k=1}^{53} f_k \times 2^{-k} \quad -1021 \leqslant e \leqslant +1024$$

float.h中对应于这些类型的值如表5-4所示。float类型的浮点常量使用标准C后缀F表示它们的类型。在标准C中，IEEE支持是可选的。

表5-4 IEEE浮点特征

名称	FLT_名称值	DBL_名称值
RADIX	2	不适用
ROUNDS	由编译器定义	不适用
EPSILON	1.19209290E-07或0x1P-23F(C99)	2.2204460492503131E-16或0x1P-52(C99)
DIG	6	15
MANT_DIG	24	53
DECIMAL_DIG[a]	17	17
MIN	1.17549435E-38F或0x1P-126F(C99)	2.2250738585072014E-308或0x1P-1022(C99)
MIN_EXP	−125	−1021
MIN_10_EXP	−37	−307
MAX	3.40282347E+38F或0x1.fffffeP127F(99)	1.7976931348623157E+308或0x1.fffffffffffffP1023(C99)
MAX_EXP	128	1024
MAX_10_EXP	38	308

a 这个名称的前面没有FLT_或DBL_前缀。

参考：浮点常量 第2.7.2节；浮点转换 第6.2.4节；浮点表示形式 第6.1.1；NaN相关的函数 第17.14节、第17.15节。

复数浮点类型

C99在C中增加了6种复数类型：float _Complex、double _Complex、long double _Complex、float _Imaginary、double_Imaginary、long double _Imaginary。复数类型被认为是浮点类型和算术类型。非复数的算术类型称为实数类型。在C的独立实现中，并不需要实现任何复数类型。即使在宿主实现中，纯虚数_Imaginary类型也是可选的。

复数类型指定符： (C99)

```
float _Complex
double _Complex
long double _Complex
```

选择关键字_Complex是为了避免与现有程序中可能存在的用户定义类型名称complex冲突。关键字_Complex前面的类型指定符指定了对应的实数类型。头文件complex.h定义了complex宏，它与_Complex同义。因此，不使用遗留系统的程序员可以使用这个更简单的名称。

每种复数类型都用一个包含2个元素的对应实数类型的数组表示，并具有与数组相同的对齐要求。第1个元素表示这个复数的实数部分，第2个元素表示虚数部分。

C99编译器可以选择支持纯虚数类型float _Imaginary、double _Imaginary和long double _Imaginary。它们也被认为是复数类型，但它们用1个对应实数类型的元素表示。它们可以方便地在各种类型的复数计算中使用，但还不足以成为标准的官方部分。

至少有1个部分为无限的复数（或虚数）值被认为是无限的，即使它的另一部分是NaN。为了使一个复数是有限的，它的两个部分都必须是有限的（不为无限或NaN）。如果一个复数（或虚数）的两个部分都是零，则它的值就是零。

参考：复数转换 第6.2.4节；complex.h头文件 第23章；寻常双目转换 第6.3.4节。

5.3 指针类型

对于任何类型T，都可以形成指针类型“指向T的指针”。取决于T是对象还是函数，指针类型又可以分为对象指针或函数指针。指针类型的值是T类型的对象或函数的地址。第4.5.2节讨论了指针类型的声明。

例子

```
int *ip; /* ip: a pointer to an object of type int */
char *cp; /* cp: a pointer to an object of type char */
int (*fp)(); /* fp: a pointer to a function returning
                an integer */
```

指针有两个最常用的操作符。一个是取地址操作符&，用于创建指针值；另一个是间接访问操作符*，它对指针进行解引用，以访问它所指向的对象。

例子

在下面这个例子中，指针ip被赋值为变量i的地址（&i）。在这个赋值之后，表达式*ip所表示的对象与i相同：

```
int i, j, *ip;
ip = &i;
i = 22;
j = *ip;     /* j now has the value 22 */
*ip = 17;    /* i now has the value 17 */
```

指针类型的其他操作包括赋值、减法、关系和相等测试、逻辑AND和OR、与整数的加法和减法以及整数类型和指针类型之间的相互转换。

指针的长度因编译器而异。在有些情况下，指针的长度因它所指向的对象类型而异。例如，数据指针可能比函数指针更长或更短（第6.1.5节）。指针的长度并不一定和任何类型的整数长度存在任何关系，但一般可以认为long类型至少与任何指针类型一样长。在C99中，intptr_t类型表示指针的长度。

在标准C中，指针类型的前面可以使用类型限定符const、volatile和restrict（C99）。指针类型的限定（如果有）可能会影响指针的操作和转换，并因此对它进行优化。

参考：取地址操作符& 第7.5.6节；数组和指针 第5.4.1节；赋值操作符 第7.9节；类型转换表达式 第7.5.1节；指针的转换 第6.2.7节；if语句 第8.5节；间接访问操作符* 第7.5.7节；intptr_t 第21.5节；指针声明器 第4.5.2节；类型限定符 第4.4.3节。

5.3.1 通用指针

在一般编程中，偶尔需要一种通用的数据指针，可以把它转换为任何对象指针类型。在传统C中，习惯用char *来实现这个目的，在解引用之前把通用指针转换为适当的类型，具体的细节将在第6.2节描述指针转换时再介绍。使用char *所存在的问题是编译器无法检查程序员是否对指针类型进行了正确的转换。

标准C引入了void *类型，表示“通用指针”。为了与旧式的编译器兼容，它的表示形式与char *类型相同，但是C语言把它们当成不同的东西。通用指针无法用*或下标操作符进行解引用，也不能作为加法或减法操作符的操作数。任何指向对象或不完整类型的指针（不指向函数类型）都可以转换为void *类型，然后再转换回来，而不会发生任何改变。void *类型既不是对象指针，也不是函数指针。

例子

下面是一些指针声明和转换的例子：

```
void *generic_ptr;
int *int_ptr;
char *char_ptr;
generic_ptr = int_ptr;       /* OK */
int_ptr = generic_ptr;       /* OK */
int_ptr = char_ptr;          /* Invalid in Standard C */
int_ptr = (int *) char_ptr; /* OK */
```

通用指针在使用函数原型方面提供了额外的灵活性。当一个函数具有一个可以接受任何类型的数据指针的形式参数时，这个形式参数就应该被声明为void *类型。如果这个形式参数被声明为任何其他指针类型，对应的实际参数必须是相同的类型，因为不同的指针类型在标准C中并不是赋值兼容的。

例子

strcpy工具函数用于复数字符串，因此需要char *类型的实参：

```
char *strcpy(char *s1, const char *s2);
```

但是，memcpy函数可以接受任何类型的指针，所以使用void *：

```
void *memcpy(void *s1, const void *s2, size_t n);
```

参考：赋值兼容 第6.3.2节；const类型指定符 第4.4节；memcpy工具函数 第14.3节；strcpy工具函数 第13.3节。

5.3.2 null指针和非法指针

C的每种指针类型都有一个特殊的值，称为null指针，它与这种类型的每个合法指针都不相同，它等于null指针常量，并且可以转换为其他指针类型的null指针。当它用于布尔上下文环境时，它的值是“假”。C的null指针常量是值为0的任何整数表达式（或者转换为void *之后）。根据传统，NULL宏被定义为null指针常量。在标准C中，这个宏是在stddef.h头文件中定义的。在旧式编译器中，这个宏是在stdio.h文件中定义的。

通常，所有null指针的表示形式是所有的位为0，但这并非强制要求。事实上，不同的指针类型可能采用不同的null指针表示形式。如果null指针没有被表示为0，编译器必须花点心思，确保在不同类型的null指针和null指针常量之间进行正确的转换。

例子

下面这条语句：

```
if (ip) i = *ip;
```

是下面这条语句的常用简便记法：

```
if (ip != NULL) i = *ip;
```

作为一种良好的编程风格，当一个指针不再指向一个合法的对象或函数时，最好把它的值设置为NULL。

程序员也可能因为不小心而创建了非法指针，即指针的值不是NULL，但并没有指向适当的对象或函数。在声明指针变量之后，如果没有把它初始化为某个合法的指针或NULL，常常会导致非法指针。对非法指针的任何使用，包括把它与NULL值进行比较、把它作为实参传递给函数、把它的值赋值给另一个指针，其结果在标准C中是未定义的。其他可能出现非法指针的情况包括把任何整数值强制转换为指针类型、销毁指针所指向对象的存储空间（通过工具函数free），或者使用指针运算导致指针所指向的元素超出了数组的边界。试图对非法指针进行解引用将会导致运行时错误。

在使用指针运算时，C要求数组最后一个元素之后的那个位置仍然是一个可定义的地址，尽管对这种地址进行解引用可能是非法的。这样，程序员可以很方便地使用指针表达式对数组进行遍历。

例子

下面这个循环使用了数组尾部之后的地址，不过它并没有对这个地址进行解引用：

```
int array[N]; /* last object address is &array[N-1] */
int *p;
…
for (p = &array[0]; p < &array[N]; p++)
    …
```

这个要求可能会限制C编译器在一些采用非连续地址体系结构的目标计算机上的实现，只能把数组的最大长度减少1个对象。在这种计算机上，对位于非连续内存区域中的指针进行运算是不可能的。为了执行指针运算，程序员在分配内存时必须保证它是连续的。

参考：free 第16.1节；整数常量 第2.7.1节；指针运算 第7.6.2节；stddef.h工具 第11.1节；void *类型 第5.3.1节。

5.3.3 使用指针的一些警告

许多C程序假设所有的指针类型（实际上就是所有的地址）具有统一的表示形式。在常见的字节编址的计算机上，所有的指针都采用简单的字节地址，占据一定的空间（例如1个字）。在这些计算机上，指针和整数类型的转换在表示形式上不需要修改，并且不会丢失信息。

事实上，C语言并不要求这个良好的行为。第6.1节将更详细地讨论这个问题，这里只是简单地进行总结：

1. 指针常常和int类型具有不同的长度，有时候和long类型也具有不同的长度。有时候，指针的长度是个编译器选项。在C99中，intptr_t类型是个整数类型，足以容纳一个对象指针。

2. 字符指针和void *指针可以比其他类型的指针更大，并且可以采用与其他类型的指针不同

的表示形式。例如，它们可以使用在其他类型的指针中一般为0的高端位。

3. 函数指针和数据指针可能具有截然不同的表示形式，包括不同的长度。

在指针类型之间进行转换时，程序员总是应该使用显式的强制类型转换，并特别需要注意传递给函数的指针实参的类型与函数所期望的类型兼容。在标准C中，void *可以作为通用的对象指针。但是，并不存在通用的函数指针。

参考：类型转换 第7.5.1节；intptr_t 第21.5节；malloc函数 第16.1节；指针转换 第6.2.7节。

5.4 数组类型

如果T是除了void、不完整类型以及函数类型之外的任何C类型，就可以声明“T的数组”这种类型。这个类型的值是一系列的T类型的元素。所有的数组都是从0开始计数的。请参见第4.5.3节对数组声明器的语法及其含义的讨论，包括不完整数组和可变长度的数组类型。

例子

一个数组声明为int A[3]，它由元素A[0]、A[1]和A[2]组成。在下面的代码中，声明了一个整型数组（ints）和一个指针数组（ptrs），ptrs中的每个指针被设置为ints数组中对应整数的地址：

```
int ints[10], *ptrs[10], i;
for (i = 0; i < 10; i++)
   ptrs[i] = &ints[i];
```

数组所占据的内存大小（使用sizeof操作符所得到的结果）总是等于数组的元素数量乘以每个元素在内存中的大小。

参考：数组声明器 第4.5.3节；sizeof操作符 第7.5.2节；存储单元 第6.1.1节；结构类型 第5.6节；可变长度的数组 第5.4.5节。

5.4.1 数组和指针

在C中，“T的数组”和“指向T的指针”之间存在密切的关系。首先，当一个数组标识符出现在一个表达式中时，这个标识符的类型就从“T的数组”转换为“指向T的指针”，并且这个标识符的值被转换为指向数组第1个元素的指针。这是寻常单目转换规则之一。这个转换规则的唯一例外是当数组标识符作为sizeof或取地址（&）操作符的操作数时。此时，sizeof返回整个数组的长度，&返回指向数组的指针（而不是指向数组的第1个元素的指针的指针）。

例子

在下面的第2行中，a的值被转换为指向数组第1个元素的指针：

```
int a[10], *ip;
ip = a;
```

它和下面的写法完全相同：

```
ip = &a[0];
```

sizeof(a)的值将是sizeof(int) * 10，而不是sizeof(int *)。

其次，数组的下标是根据指针运算定义的。也就是说，表达式a[i]被定义为与*((a) + (i))相同，其中a根据寻常单目转换规则转换为&a[0]。下标的这个定义还意味着a[i]与i[a]相同，并且任何指针都可以像数组一样用下标访问。程序员必须保证指针指向了正确的数组元素。

例子

如果d的类型是double，dp是指向double对象的指针，考虑下面这个表达式：

```
d = dp[4];
```

只有当dp当前指向一个double数组的一个元素时，并且dp所指向的元素后面至少还有4个元素时，这个定义才是合法的。

参考：取地址操作符& 第7.5.6节；加法操作符+ 第7.6.2节；数组声明器 第4.5.3节；数组的转换 第6.3.3节；间接访问操作符 * 第7.5.7节；指针类型 第5.3节；sizeof操作符 第5.4.4节、第7.5.2节；下标 第7.4.1节；寻常单目转换 第6.3.3节。

5.4.2 多维数组

多维数组被声明为“数组的数组”。例如，在下面这个声明中：

```
int matrix[12][10];
```

matrix声明为包含12 × 10个int类型的元素的数组。C语言对数组的维数并没有施加限制。matrix数组也可以通过两个步骤来声明，使它的结构显得更加清晰：

```
typedef int vector[10];
vector matrix[12];
```

也就是说，matrix是个包含12个元素的数组，其中每个元素都是一个包含10个int类型的元素的数组。matrix的类型是int [12][10]。多维数组是按照行主序的顺序存储的。也就是说，那些只有最后一个下标不同的元素是相邻存储的。

多维数组可以转换为指针数组，就像一维数组可以转换为指针一样。

例子

数组int t[2][3]的元素是按照下面的方式存储的（按递增的地址）：

```
t[0][0], t[0][1], t[0][2], t[1][0], t[1][1], t[1][2]
```

表达式t[1][2]展开后的结果是*(*(t+1)+2)，它是按照下面的步骤进行求值的：

1. 表达式t是个2 × 3的数组，它被转换为一个指向包含这个数组前3个元素的子数组的指针。
2. 然后，表达式t+1是个指向第2个包含3个元素的子数组的指针。
3. 表达式*(t+1)，第2个包含3个整数元素的子数组被转换为一个指向该子数组的第1个元素的指针。
4. 表达式*(t+1)+2是指向第2个子数组的第3个整数元素的指针。
5. 最后，*(*(t+1)+2)是第2个子数组的第3个整数元素的值，相当于t[1][2]。

一般而言，任何“i × j × … × k的T类型的数组”类型的表达式A可以立即转换为“指向i × j × … × k的T类型的数组的指针”。

参考：加法操作符+ 第7.6.2节；数组声明器 第4.5.3节；间接访问操作符* 第7.5.7节；指针类型 第5.3节；下标 第7.4.1节。

5.4.3 数组边界

在分配了数组的存储空间之后，数组的长度必须是已知的。但是，由于编译器在正常情况下并不会检查下标是否位于被声明的数组的边界之内，因此在声明外部一维数组（在另一个模

块中定义）或者声明作为函数形参的一维数组时，可以省略数组的长度（参见第4.5.3节）。

例子

下面这个函数sum返回外部数组a的前n个元素的和。它并没有指定数组a的边界：

```
extern int a[];

int sum(int n)
{
    int i, s = 0;
    for (i = 0; i < n; i++)
        s += a[i];
    return s;
}
```

这个数组也可以像下面这样作为函数的形参：

```
int sum(int a[], int n)
{
    int i, s = 0;
    for (i = 0; i < n; i++)
        s += a[i];
    return s;
}
```

形参a可以声明为int *a，而不必对函数体进行任何修改。它更精确地反映了实现细节，但对函数意图的说明则不如前者。

在使用多维数组时，必须要指定除了第1维之外其他各维的边界，以便进行正确的地址计算：

```
extern int matrix[][10]; /* ?-by-10 array of int */
```

如果未指定边界，这个声明就是错误的。在C99中，数组可以具有可变长度，包括多维数组。

参考：数组声明器 第4.5.3节；定义性声明和引用性声明 第4.8节；间接访问操作符* 第7.5.7节；省略数组边界 第4.5节；指针类型 第5.3节；下标 第7.4.1节；可变长度的数组 第5.4.5节。

5.4.4 操作

可以直接应用于数组的操作符只有sizeof操作符和取地址操作符&。当数值作为sizeof的操作数时，它必须具有边界，其结果是这个数组所占据的存储单元的数量。对于包含n个元素的T类型的数组，sizeof的结果总是等于n乘以sizeof T。&操作符的结果是个指向数组（数组第1个元素）的指针。

在其他上下文环境中，例如对数组进行下标操作时，数组实际上是作为一个指针使用的，因此所有适用于指针的操作也同样适用于数组。

参考：数组声明器 第4.5.3节；数组转换为指针 第6.2.7节；指针类型 第5.3节；sizeof操作符 第7.5.2节；下标 第7.4.1节。

5.4.5 可变长度的数组

C99向C程序员提供了可变长度数组的功能。所谓可变长度的数组，就是数组的长度在运行时才能确定的数组。可变长度数组的声明形式与固定长度的数组相似，唯一的区别是数组的长度是由一个非常量表达式指定的。当编译器遇到这样的声明时，将对长度表达式进行求值，并

用它的结果（必须是个正整数）作为数组的长度，创建这个数组。在创建之后，可变长度的数组就不能再改变长度。数组中的元素只能在已分配的空间内访问。访问这个空间之外的元素将产生未定义的行为。但是，编译器并不要求对这种越界访问进行检查。当包含数组声明的代码块结束时，这个数组就被销毁。每次进入这个代码块时，将会创建一个新的数组。

除了作为函数的数组形参之外，可变长度的数组必须在代码块作用域中声明，并且必须不能声明为static或extern。只有普通的标识符（而不是结构或联合成员）可以被声明为可变长度的数组。数组变量的作用域从数组的声明位置一直到包含数组声明的最内层代码块的结束。数组的生命期从数组被声明开始一直到程序的执行离开这个数组的作用域。在C99中，离开作用域的情况包括完成代码块的执行、跳出代码块或者跳回到代码块中这个声明之前的位置。编译器在处理这类数组声明时可以在执行堆栈上为它们分配空间。

可变修改类型（variably modified type）包括可变长度的数组以及把可变长度的数组作为自身一部分的其他类型，例如指向可变长度数组的指针。只有具有代码块作用域并且没有链接属性的普通标识符可以被声明为可变修改类型。这里就留下了一个漏洞：可以把一种可变修改类型（但不是可变长度的数组）作为一个具有代码块作用域的static标识符的类型。在这种情况下，尽管这个static标识符的值在这个代码块每次完成执行之后仍然被保留，但这个标识符所包含的可变长度数组在每次进入这个代码块时都可能改变长度。

例子

在下面的代码片段中，a和b都是可变长度的数组，指针c是一种可变修改类型。

```
int a_size;
...
void f(int b_size)
{
    int c_size = b_size + a_size;
    int a[a_size++];
    int b[b_size][b_size];
    static int (*c)[5][c_size];
    ...
}
```

对可变长度数组的这些限制简化了C99的实现，而且仍然保留了它们的绝大多数用途。如果没有这个限制，就会出现大量的复杂性和交互性。例如，必须为结构提供一种隐藏类型描述机制，以便声明可变修改类型的成员。在声明具有文件作用域的可变长度数组时，不得不付出在运行时“详细分析”顶层声明的开销（C++和其他语言提供了这种机制，但它们不符合C的精神）。

如果在一个typedef声明中使用一个可变长度的数组，它的长度表达式只求值一次，也就是在进入这个typedef声明时，而不是在每次使用这个新类型名称时。

例子

```
/* Assume n has the value 5 here */
typedef int[n] vector;
n += 1;
vector a;
int b[n];
```

变量a是个包含5个整数元素的数组，反映了遇到这个typedef声明时n的值。相比之下，b是一个包含6个整数元素的数组，因为在遇到b的声明时，n的值已经发生了变化。

可变长度数组形参 可变长度数组或可变修改类型可以作为函数的形参类型。当数组的长度也是个形参时，它必须首先出现，这是C的词法作用域规则所决定的。

当一个具有可变长度数组形参的函数被调用时，传递给这个函数的数组实参的每一维的长度必须与函数定义中这个数组形参声明一致，否则结果就是未定义的。

例子

第1个函数定义是正确的。第2个函数定义可能是非法的，也可能通过其他变量r和c计算a的各维的长度。

```
void f( int r, int c, int a[c][r] ) {...} /* OK */
void f( int a[c][r], int r, int c ) {...}
   /* NO: r, c are not visible to a[c][r] */
```

在函数原型声明中（不是函数定义的一部分），可变长度数组各维的长度可以由[*]指定。在函数原型中，出现在数组方括号内的任何非常量表达式都被当做[*]。

例子

下面这些原型都是兼容的。尽管第3个原型提示了一个正方形的数组，但编译器并不会在编译时检查这个限制。最后一个原型说明了可以简单地省略可变长度数组的第1维（也仅有此维）的长度。

```
void f(int, int [*][*]);
void f(int n, int [*][m]);
void f(int n, int [n][n]);
void f(int, int [][*]);
```

参考：数组声明器 第4.5.3节；函数原型 第9.2节；sizeof操作符 第7.5.2节。

5.5 枚举类型

下面显示了声明枚举类型的语法：

枚举类型指定符：

枚举类型定义
枚举类型引用

枚举类型定义：

enum 枚举标签$_{opt}$ { 枚举定义列表 }
enum 枚举标签$_{opt}$ { 枚举定义列表，} (C99)

枚举类型引用：

enum 枚举标签

枚举标签：

标识符

枚举定义列表：

枚举常量定义
枚举定义列表，枚举常量定义

枚举常量定义：

枚举常量

枚举常量 = 表达式

枚举常量：

标识符

在C中，枚举类型是由一种称为枚举常量（enumeration constant）的标识符所表示的一组整数值。枚举常量是在定义枚举类型时指定的，它的类型为int。每个枚举类型都是由一种因编译器而异的整数类型表示的，并且与这种类型兼容。因此，就类型检查这个目的而言，枚举类型只不过是其中一种整数类型。在某个上下文环境中，如果C语言允许使用整数表达式，它也同样允许使用枚举常量或者枚举类型的值（在C++中，情况并非如此，参见第5.13.1节）。

C99允许在枚举定义列表的最后添加一个分号，这是为了适应程序员的习惯而作出的一个小小的改进。

例子

下面这个声明

```
enum fish { trout, carp, halibut } my_fish, your_fish;
```

创建了一种枚举类型enum fish，它的值包括trout、carp和halibut。它还声明了两个这种枚举类型的变量my_fish和your_fish，它们可以使用下面的语句进行赋值：

```
my_fish = halibut;
your_fish = trout;
```

枚举类型的变量或其他对象可以在包含该枚举类型定义的声明中声明，也可以通过“枚举类型引用”，在后续的提到这种枚举类型的声明中声明。

例子

例如，下面这个声明

```
enum color { red, blue, green, mauve }
   favorite, acceptable, least_favorite;
```

相当于下面这2个声明：

```
enum color { red, blue, green, mauve } favorite;
enum color acceptable, least_favorite;
```

也相当于下面这4个声明：

```
enum color { red, blue, green, mauve };
enum color favorite;
enum color acceptable;
enum color least_favorite;
```

枚举标签color允许这个枚举类型在定义之后被引用。尽管下面这个声明

```
enum { red, blue, green, mauve }
   favorite, acceptable, least_favorite;
```

声明了相同的变量和枚举常量，但由于没有枚举标签，所以无法在以后的声明中声明这种类型的其他变量。

枚举标签的重载分类与结构标签和联合标签相同，它们的作用域与在源程序的相同位置所定义的变量相同。

声明为枚举常量的标识符与变量、函数和typedef名称属于同一个重载分类。它们的作用域

与在源程序的相同位置所定义的变量相同。

例子

在下面的代码中，作为枚举常量声明的shepherd隐藏了前面的整型变量shepherd的声明。但是，浮点型变量collie的声明却导致了编译错误，因为collie已经在同一个作用域中被声明为一个枚举常量。

```
int shepherd = 12;
{
    enum dog_breeds {shepherd, collie};
        /* Hides outer declaration of the name "shepherd" */
    float collie;
        /* Invalid redefinition of the name "collie" */
}
```

枚举类型是用与枚举常量相关联的整数值实现的，这样枚举类型的值的赋值和比较可以实现为整数的赋值和比较。整数值按照下面的方法与枚举常量相关联：

1. 通过在类型定义中使用下面的写法，显式的整数值可以与枚举常量相关联

```
枚举常量 = 表达式
```

表达式必须是个整数类型的常量表达式，包括前面定义枚举常量时所使用的表达式，例如：

```
enum boys { Bill = 10,
            John = Bill+2,
            Fred = John+2 };
```

2. 如果没有显式地指定，第1个枚举常量的默认值是0。

3. 如果没有显式地指定，后续每个枚举常量的值比它前面的那个枚举常量的值大1。

任何可以用int类型表示的有符号整数值都可以与枚举常量相关联。可以选择正整数或负整数表示枚举常量，甚至可以用同一个整数值表达两个不同的枚举常量。

例子

在下面这个声明中

```
enum sizes { small, medium=10, pretty_big, large=20 };
```

small、medium、pretty_bit和large的值分别是0、10、11和20。尽管下面这个声明也是合法的

```
enum people { john=1, mary=19, bill=-4, sheila=1 };
```

它的效果导致表达式john == sheila为真，这个结果并不是很直观。

尽管枚举类型定义的格式好像是进行了类型检查的结构和联合类型。事实上，标准C中的枚举类型（也是本书所采用的定义）只是一种使整数常量的名称更具描述性的方法。作为一种风格，我们建议程序员把枚举类型看成与整数不同，不要在整型表达式中混合使用它们（在不使用强制类型转换的情况下）。事实上，有些UNIX C编译器实现了一种弱类型形式的枚举，在不使用强制类型转换的情况下，不允许枚举类型和整数之间的转换。

参考：类型转换表达式 第7.5.1节；标识符 第2.5节；重载分类 第4.2.4节；作用域 第4.2.1节。

5.6 结构类型

C的结构类型与其他编程语言中的记录类型相似。它们是可以具有不同类型的有名称的成分

（又称字段或成员）的集合。结构可以被定义为封装了相关的数据对象。

结构类型指定符：

结构类型定义
结构类型引用

结构类型定义：

struct 结构标签$_{opt}$ { 字段列表 }

结构类型引用：

struct 结构标签

结构标签：

标识符

字段列表：

成员声明
字段列表 成员声明

成员声明：

类型指定符 成员声明器列表;

成员声明器列表：

成员声明器
成员声明器列表,成员声明器

成员声明器：

简单成员
位段

简单成员：

声明器

位段：

声明器$_{opt}$:宽度

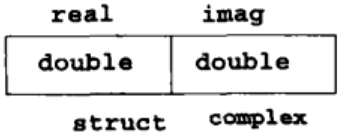
宽度：

常量表达式

例子

希望实现复数类型的程序员（在C99之前）可能会定义一种complex结构，用real和imag成员保存复数的实部和虚部。第1个声明定义了这种新类型，第2个声明则声明了两个这种类型的变量x和y：

```
struct complex {
    double real;
    double imag;
};
struct complex x,y;
```



new_complex函数用于创建这种类型的新对象。注意，选择操作符(.)用于访问结构的成员：

```
struct complex new_complex(double r, double i)
{
    struct complex c;
    c.real = r;
    c.imag = i;
    return c;
}
```

另外，可以定义在这种类型上所执行的操作，例如complex_multiply：

```
struct complex complex_multiply( struct complex a,
                                 struct complex b )
{
    struct complex product;
    product.real = a.real * b.real - a.imag * b.imag;
    product.imag = a.real * b.imag + a.imag * b.real;
    return product;
}
```

例子

下面这个声明

```
struct complex { double real, imag; } x, y;
```

相当于下面这两个声明

```
struct complex { double real, imag; };
struct complex x, y;
```

5.6.1 结构类型引用

合成类型“结构类型定义”或“联合类型定义（第5.7节）”中的类型指定符引入了一种新的类型，它与其他所有类型不同。如果定义中出现了结构标签，它就与这种新类型相关联，可以在后续的结构类型引用中使用。

结构类型定义（以及可能存在的结构标签）的作用域是从声明点直到包含这个指定符的最内层代码块的结束。新定义显式地重写（隐藏）了这个外层代码块中该类型标签的任何定义。

对于合成类型“结构类型定义”或“联合类型定义（第5.7节）”而言，即使在同一个代码块或外层代码块中还没有出现它的定义，仍然可以使用它们的类型指定符，只要此时并不需要知道结构的长度。它包括下面这两种情况：

1. 指向结构的指针。
2. 作为结构同义词的typedef名称。

使用这种类型的指定符在当前的最内层代码块中引入了这种类型和类型标签的“不完整”定义。为了使定义变得完整，结构类型定义或联合类型定义必须出现在同一作用域的后面。

作为一种特殊情况，在声明中未使用声明器的结构类型引用或联合类型引用将隐藏该类型标签在外层作用域中的任何定义，并创建了一种不完整类型。

例子

考虑一个内层代码块中两个自引用的结构的正确定义：

```
{
    struct cell;
```

```
    struct header { struct cell   *first; … };
    struct cell   { struct header *head;  … };
    …
}
```

第1行的不完整定义“struct cell”隐藏了外层代码块中的cell标签的任何定义。第2行的定义struct header自动隐藏了外层代码块中header的任何定义。另外，它使用struct cell定义一个指针是合法的。第3行的struct cell定义提供了cell的完整信息。

不完整类型声明还存在于结构类型定义或联合类型定义中，从新标签的第1次出现直到定义结束为止。这就允许结构包含一个指向自身的指针作为它的成员（见图5-1）。

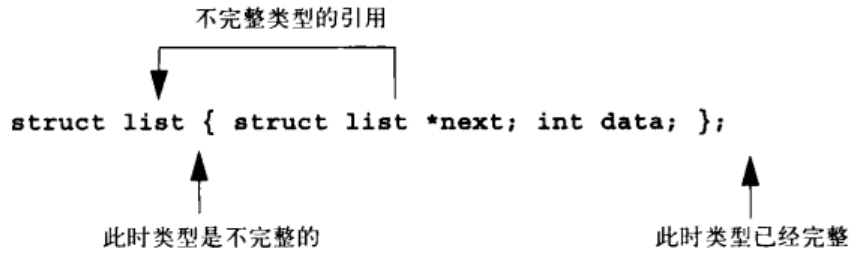


图5-1 声明中不完整的结构类型

参考：声明 第4.1节；声明器 第4.5节；重复可见性 第4.2.2节；作用域 第4.2.1节；选择操作符. 第7.4.2节；等价类型 第5.11节。

5.6.2 结构上的操作

为结构所提供的操作因编译器而异。所有的C编译器都提供了对结构进行操作的选择操作符.和->。现在，更新的编译器允许对结构进行赋值，把它作为参数传递给函数，以及作为函数的返回值（在较旧的编译器中，结构的赋值必须逐成员进行，并且只有指向结构的指针可以作为参数传递给函数以及从函数返回）。

比较两个结构的相等性是不允许的。一种结构类型的对象由一系列其他类型的成员组成。由于某些数据对象可能受到目标计算机在地址边界上的一些限制，因此结构对象可能会包含“空洞”，即不属于该结构的任何成员的存储单元。由于这种空洞的存在，如果把结构的比较实现为逐位的比较，无疑是不可靠的，但逐成员的比较又显得开销太大（当然，程序员可以编写对结构进行逐成员比较的函数）。

在任何允许使用单目取地址操作符&对结构进行操作以获取指向该结构的指针的场合，也允许对这个结构的成员应用&操作符，以获取指向该成员的指针。使一个指针指向结构的中间是可能的。这个规则的一个例外是当结构的成员被定义为位段的时候。被定义为位段的成员一般并不位于计算机可编址边界上，因此无法形成指向位段的指针。C语言并没有提供指向位段的指针。

参考：取地址操作符& 第7.5.6节；赋值 第7.9节；位段 第5.6.5节；相等操作符== 第7.6.5节；选择操作符.和-> 第7.4.2节；类型相等性 第5.11节。

5.6.3 成员

结构的成员可以是任何不属于可变修改的对象类型。结构不能包含自身的实例为成员，但它可以包含指向自身的实例的指针为成员。

在C99中，结构的成员不可以是可变修改类型。结构的最后一个成员可以是不完整的数组类型，在这种情况下称为灵活数组成员（第5.6.8节）。

例子

下面这个声明是非法的：

```
struct S {
    int a;
    struct S next; /* invalid! */
};
```

但下面这个声明是允许的：

```
struct S {
    int a;
    struct S *next; /* OK */
};
```

结构成员的名称是在一个特殊的重载分类中定义的，与结构的类型相关联。也就是说，在一个结构的内部，结构的成员名称必须唯一，但它们可以与其他结构的成员名称相同，也可以与变量、函数和类型名称相同。

例子

考虑下面的声明序列：

```
int x;
struct A { int x; double y; } y;
struct B { int y; double x; } z;
```

标识符x具有3个非冲突的声明，分别是个整型变量、结构A的一个整数成员以及结构B的一个浮点成员。这三个声明分别在下面的表达式中使用：

```
x
y.x
z.x
```

如果结构标签是在其中一个成员中定义的，这个标签的作用域就延续到包含这个结构定义的代码块的结束（如果这个结构是在顶层定义的，则这个内层标签的作用域也是如此）。

例子

在下面这个声明中：

```
struct S {
    struct T {int a, b; } x;
};
```

标签T的定义范围从它的第1次出现直到定义S的那个作用域的结束。

历史说明：C的最初定义指定了所有结构中的所有成员是在相同的重载分类中分配存储空间的，因此两个结构不能拥有相同名称的成员（当不同结构中的成员具有相同的类型和相同的相对位置时，就是出现异常）。这似乎有些不可思议，但有些旧式的文档就是这样说明的，并且有些旧式的编译器确实是这样实现的。

参考：灵活数组成员 第5.6.8节；不完整数组类型 第5.4节；重载分类 第4.2.4节；作用域 第4.2.1节；可变修改类型 第5.4.5节。

5.6.4 结构成员的布局

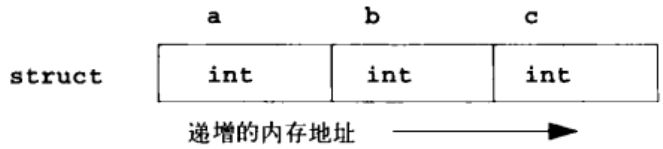
大多数程序员并不关心结构的成员是如何包装的。但是，C向程序员提供了一些对结构成员的包装进行控制的能力。C编译器必须以严格的顺序按照递增的内存地址为结构成员分配空间，第1个成员从结构的起始地址开始。

例子

下面这两个结构在成员布局方面并没有什么区别：

```
struct { int a, b, c };
struct { int a, int b, c; };
```

以下图所示的次序摆放a、b和c。



假设有两个指针p和q，它们分别指向同一个结构中的成员。当且只当结构声明中p所指向的成员的声明出现在q所指向的成员的声明之前时，p < q才是真的。

例子

```
struct vector3 { int x, y, z; } s;
int *p, *q, *r;
...
p = &s.x;
q = &s.y;
r = &s.z; /* At this point p < q, q < r, and p < r. */
```

在结构的布局中，两个连续的成员之间或者最后一个成员之后可能会出现空洞或填充，以实现成员的正确对齐。这种空洞的位模式是不可预测的，不同的结构会不同，即使在同一个结构中，随着时间的变化，它们的位模式也可能不同。填充所占据的空间包含在sizeof操作符所返回的值中。有些编译器提供了设置和开关，用于控制结构的包装方式。

5.6.5 位段

C允许程序员把整数成员包装到比编译器正常所允许的更小的空间中。这种整数成分称为位段，是通过在成员声明器中使用一个冒号和一个常量表达式（指定了位段所占据的位数）指定的。

例子

下面这个结构具有3个成员a、b和c，它们分别占据4、5和7个位：

```
struct S {
    unsigned a:4;
    unsigned b:5, c:7;
};
```

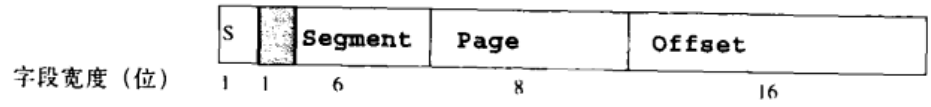
n位的位段可以表示0到2^n-1范围之内的无符号整数，或者-2^{n-1}到$2^{n-1}-1$之间的有符号整数（如果有符号整数采用补码表示形式）。C的最初定义只允许unsigned类型的位段，但标准C允许位段为unsigned int、signed int或简单的int类型，它们分别称为无符号位段、有符号位段和普通位段。和普通字符一样，普通位段可能是有符号或无符号的。有些C编译器允许使用任何整数类

型的位段，包括char类型。C99还允许_Bool类型的位段。

位段一般用于依赖机器的程序，这种程序必须强制数据结构对应于一种固定的硬件表示形式。在结构中包装成员（尤其是位段）的精确行为因机器而异，但在每种计算机中，这种行为是可以预测的。它的意图是使位段根据本节后面所讨论的规则，尽可能紧密地包装在结构中。因此，使用位段的程序是无法移植的。程序员应该查阅机器文档，看看是否需要按照某些特定的方式设置结构在内存中的布局，并验证C编译器确实按照预期的方式对结构的成员进行了包装。

例子

下面是如何使用位段创建与一个预定义的格式匹配的结构的例子。下图是一个32位的字的布局，它被看成是一台假想的计算机上的一个虚拟地址。这个字包含了表示段号、页号和页内偏移值的字段，另外还有1个“管理员”位和1个未使用的位。



为了复制这个布局，我们首先必须知道计算机是从左到右还是从右到左包装位段的。也就是说，计算机所采用的是“大端字节顺序”还是“小端字节顺序”（参见第6.1.2）。如果包装顺序是从右到左，正确的结构定义是：

```
typedef struct {
   unsigned Offset     : 16;
   unsigned Page       : 8;
   unsigned Segment    : 6;
   unsigned UNUSED     : 1;
   unsigned Supervisor : 1;
} virtual_address;
```

反之，如果包装顺序是从左到右，正确的结构定义是：

```
typedef struct {
   unsigned Supervisor : 1;
   unsigned UNUSED     : 1;
   unsigned Segment    : 6;
   unsigned Page       : 8;
   unsigned Offset     : 16;
} virtual_address;
```

普通整数位段的符号遵循普通字段的符号规则。也就是说，一个普通的整数位段实际上可能被实现为有符号或无符号类型（见第5.1.3节）。有符号和无符号位段必须分别被实现为保存有符号和无符号值。

例子

考虑下面这些标准C声明在一台采用补码表示形式的计算机上的效果：

```
struct S { unsigned ubf:3;
           signed sbf:3;
           int bf:3; } x = { -1, -1, -1 };
...
int i = x.ubf;
int j = x.sbf;
int k = x.bf;
```

i的值必须是7，j的值必须是−1，但k的值可能是7，也可能是−1。

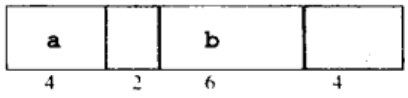
编译器可以选择地对位段的最大长度施加限制，并指定位段无法跨越的地址边界。这些对齐限制通常与目标计算机的自然字长有关。当一个字段对于一台计算机而言太长时，编译器就会产生一条适当的错误消息。当一个字段将跨越一个字的边界时，它可能会移动到下一个字。

结构中也可以包含无名位段，作为相邻成员之间的填充。无名位段无法被引用，它们的内容在运行时是不可预测的。

例子

下面这个结构把成员a放在结构的前4个位中，它的后面是2个位的填充，再接着是占据6个位的成员b（假设基本的字长是16位，这个结构的最后4个位未使用，参见第5.6.7节）。

```
struct S {
    unsigned a : 4;
    unsigned   : 2;
    unsigned b : 6;
};
```



为一个无名字段指定0的长度具有特殊的含义，它表示不应该有其他位段被包装到前一个位段所在的区域。如果有，它就被放置。这里所指的区域是指一些因机器而异的存储单位。

例子

在下面这个结构中，成员b应该在成员a之后的一个自然地址边界（例如16位）开始。这个新结构所占据的存储空间两倍于原来那个结构：

```
struct S {
    unsigned a : 4;
    unsigned   : 0;
    unsigned b : 6;
};
```



取地址操作符&可能无法应用于位段成员，因为许多计算机无法直接对任意长度的字段进行编址。

参考：取地址操作符& 第7.5.6节；对齐限制 第6.1.3节；_Bool类型 第5.1.4节；字节顺序 第6.1.2节；枚举类型 第5.5节；有符号类型 第5.1.1节；无符号类型 第5.1.2节。

5.6.6 移植性问题

由于一些原因，依赖字节包装策略是危险的。首先，计算机对于不同的数据类型具有不同的对齐限制。例如，4字节的整数在有些计算机上可能从4的倍数的字节边界开始，但在其他计算机上可能对齐于最邻近的字节边界。

其次，位段宽度的限制也存在不同。有些计算机使用16位的字长，这就限制了位段的最大长度，并施加了位段无法跨越的边界限制。其他计算机则使用32位的字长，因此具有不同的限制。

第三，不同的计算机可能采用不同的方式把位段包装到字中，也就是它们的“字节顺序”。在Motorola 68000系列计算机中，字符是从左到右包装到字中的，从最高位到最低位。在Intel 80x86计算机上，字符是从右向左包装的，从最低位到最高位。正如前一节的virtual_address例子所示，在不同字节顺序的计算机上需要使用不同的结构定义。

就我们所知，在两种情况下适合使用位段：

1. 必须准确地匹配预定义的数据，使它可以在C程序中被引用（这种程序可能无法被移植）。

2. 必须维护一个包含结构化数据的数组。由于数组太大，所以要求结构的成员必须紧密包装，以节省内存。

通过使用C的位操作符进行掩码和移位，可以在字的内部实现对字节顺序不敏感的位段。

例子

考虑在前面的virtual_address结构中访问Page字段的例子。由于这个8位的位段位于一个字的低16位，因此它可以通过下面的代码进行访问：

```
unsigned V;     /* formatted as a virtual_address */
int Page;
...
Page = (V & 0xFF0000) >> 16;
```

它相当于可读性更佳的结构成员访问方式Page = V.Page，但是掩码和移位方法和virtual_address定义中的计算机字节顺序无关。下面的V==0xb393352e(Page==0x93)演示了掩码和移位操作：

```
10110011100100110011010100101110   V
00000000111111110000000000000000   0xFF0000
00000000100100110000000000000000   V & 0xFF0000
00000000000000000000000010010011   (V & 0xFF0000)>>16
```

类似的方法可以用于设置位段的值。这两种访问方法的运行时效率差别很小。

参考：对齐限制 第6.1.3节；位操作符 第7.6.6节；字节顺序 第6.1.2节；移位操作符 第7.6.3节。

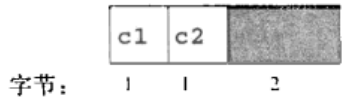
5.6.7 结构的大小

结构类型的对象的大小就是表示这种类型的所有成员所需要的存储空间的数量，包括成员之间或之后所有未使用的填充空间。规则是结构将进行填充，直到它所占据的空间等于该类型的数组的一个元素所占据的空间（对于包括结构在内的任何类型T，T类型的n个元素的数组的大小等于T的大小乘以n）。另一种表述方法是结构的结束地址必须与起始地址具有相同的对齐要求。也就是说，如果结构必须从偶数的字符边界开始，它也必须以偶数的字节边界结束。结构类型的对齐要求至少与它的对齐要求最严格的那个成员同样严格。

例子

在一台所有的结构地址都是从4字节的倍数开始的计算机上，下面这个结构的长度将是4的倍数（可能正好为4），尽管实际上所使用的只有2个字节。

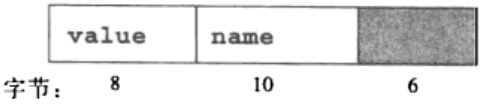
```
struct S {
    char c1;
    char c2;
};
```



例子

在一台要求所有double类型的对象的地址都是8字节的倍数的计算机上，下面这个结构的长度可能是24，尽管它在声明中只使用了18个字节。

```
struct S {
    double value;
    char name [10];
};
```

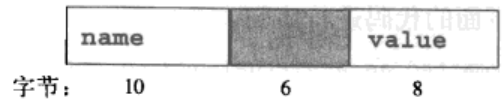


尾部6个额外的填充单位使这个结构的大小正好是对齐要求（8）的倍数。如果未使用填充空间，在这种结构类型的数组中，并不是所有的元素都正确地按照8个字节的地址对齐。

例子

对齐要求可能导致在结构的中间出现填充字节。在前面这个例子中，如果成员的顺序相反，这个结构的长度仍然是24，但未使用的空间出现在两个成员的中间，使value成员可以相对于结构的起始地址按照8字节倍数的地址对齐：

```
struct S {
    char name [10];
    double value;
};
```



这个结构类型的任何对象需要以8的倍数为起始地址，使该对象的value成员始终能够正确地对齐。

5.6.8 灵活数组成员

在C99中，结构的最后一个成员可以是个不完整的数组类型，在这种情况下它被称为灵活数组成员。引入灵活数组成员是为了使一种长期使用但并不安全的C编程惯用法合法化。这种惯用法就是长度在运行时可以改变的结构。

为了使用灵活数组成员，声明一个结构类型S，它的最后一个成员是个元素类型为E的灵活数组成员F。类型S不能只包含F，它至少必须包含另一个有名称的成员。例如：

```
struct S { int F_len; double F[]; }; /* E is double */
```

sizeof(S)的值被定义为忽略了成员F之外这个结构的长度，除掉为了实现对齐在F之前必须包含的填充字节（为了确定所需要的填充字节，可以假设F被声明为一个和原来类型相同的固定长度的数组，并在F之前使用所需的填充字节）。

通过一个S类型的左值访问一个数据对象时，必须把F看成是一个长度为L的固定长度的数组，使S不会超过这个数据对象的长度。也就是说，如果这个数据对象的长度为D，L就是满足sizeof(S) + L * sizeof(F) <= D的最大非负整数，这样就可以访问F[0]、F[1]、…、F[L-1]。如果只是简单地声明一个S类型的变量，就无法使用这个数组，因为这个数据对象（变量）并不包含这个数组的空间（D的长度等于sizeof(S)，因此L将是0）。但是，即使这个数组并无空间，仍然可以引用&F[0]。

为了使用类型S使用大于它本身的数据对象，可以声明一个指向S的指针，并把它赋值为一个更大对象的地址，或者可以使用一个联合，在一个更大的对象上重叠S。

例子

使用灵活数组成员定义一个结构，使它包含一个可变长度的向量以及该向量的长度是一种常用的做法。

```
struct Vec { int len; double vec[]; }
```

如果向量的长度是个常量，可以采用静态的方式声明。

```
#define N 20 /* Length of vector */
union{
    char data_object[sizeof(struct Vec) + N*sizeof(double)];
    struct S v;
```

```
} u = { .v = {N} }; /* C99 designated initializer */
```

如果向量的长度直到运行时才可知，可以使用malloc为它分配空间。

```
struct Vec *p;
int n; /* length of vector */
...
p = malloc( sizeof(struct Vec) + n * sizeof (double));
p->len = n;
```

下面是程序员可能使用向量的方式。

```
for (i = 0; i < u.v.len; i++) u.v.vec[i] = 0.0;
for (i = 0; i < p->len; i++) p->vec[i] = 0.0;
```

在C99之前，必须声明一个像下面这样的结构：

```
struct Vec { int len; double vec[1]; }
```

并把对malloc的调用修改为像下面这样：

```
p = malloc( sizeof(struct Vec) + (len -1)*sizeof (double));
```

尽管这样的代码通常可以完成任务，但它的行为是（并且仍然是）未定义的。

5.7 联合类型

定义联合类型的语法几乎与结构类型相同。

联合类型指定符：

联合类型定义
联合类型引用

联合类型定义：

union 联合标签$_{opt}$ {字段列表}

联合类型引用：

union 联合标签

联合标签：

标识符

定义联合成员的语法也与结构成员相同。在传统C中，联合无法包含位段，但标准C取消了这个限制。

与结构和枚举一样，每个联合类型定义引入了一个新的、与众不同的联合类型标识符。如果联合定义中存在联合标签，它就与这种新的联合类型相关联，并可以在后续的联合类型引用中使用。联合类型也允许使用前向引用和不完整定义，其规则和结构类型相同。

联合的成员可以是不属于可变修改类型的任何对象。另外，联合不能包含自身的实例，不过可以包含指向自身的指针。和结构一样，联合成员的名称是在一个特殊的与该联合类型相关联的重载分类中定义的。也就是说，一个联合内部各成员的名称必须各不相同，但它们可以与其他联合的成员同名，并可以与变量、函数和类型使用相同的名称。

5.7.1 联合成员的布局

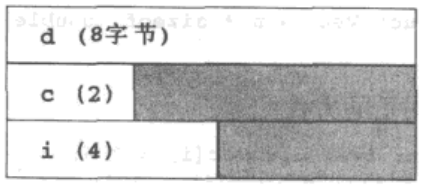
联合类型的每个成员所分配的存储空间是从该联合的起始地址开始的。一个联合在任一时

刻只能包含它的1个成员的值。联合类型的对象的起始地址总是满足它所包含的任何成员的存储对齐要求。

例子

下面是一个具有3个成员的联合，它们都有效地重叠于内存中：

```
union U {
    double d;
    char c[2];
    int i;
};
```



例子

如果已有下面的联合类型和对象定义：

```
static union U {...; int C; ...; } object, *P = &object;
```

那么下面这两个表达式具有相同的内容：

```
(union U *) & (P->C) == P
&(P->C) == (int *) P
```

而且，它们所包含的值总是相同的，不管成员C的类型是什么，也不管C之前或之后的其他成员的类型是什么。

参考：对齐限制 第6.1.3节。

5.7.2 联合的大小

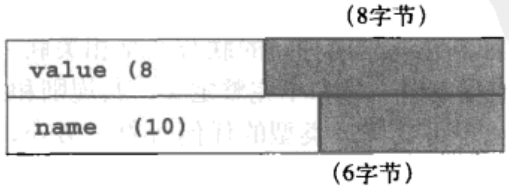
联合类型对象的长度就是表示它的最大成员的值所需要的存储空间，以及尾部的必要填充空间，以满足适当的边界对齐要求。规则是联合将进行填充，直到它所占据的空间等于该类型的数组的一个元素所占据的空间。记住，对于任何类型T（包括联合），包含n个元素的T类型的数组的大小等于T的大小乘以n。另一种说法是联合的终止地址和起始地址具有相同的对齐要求。也就是说，如果一个联合必须以偶数字节的地址开始，它也必须以偶数字节的地址结束。

注意，联合类型的对齐要求至少与它的对齐要求最严格的那个成员同样严格。

例子

在一台要求所有double类型的对象的地址为8的倍数的计算机上，下面这个联合的长度将是16，尽管最大成员的长度只有10：

```
union U {
    double value;
    char name [10];
};
```



它需要6个额外的填充字节，使这个联合符合对齐要求，也就是8的倍数。如果未使用填充字节，在一个该联合类型的数组中，并不是所有的联合元素的value成员的值都能正确地满足8的倍数这个对齐要求。

5.7.3 使用联合类型

C的联合类型有点像其他语言的“可变记录”。和结构一样，联合被定义为包含了一些成员。但是，和结构不同，联合在任一时刻最多只能容纳它的1个成员。从概念上说，联合的所有成员在该联合的存储空间中是重合的。如果这个联合非常大，或者需要一个包含大量联合对象的数组，就可以显著地节省内存。

例子

假设我们希望一个对象根据情况可以是个整数或浮点数。我们定义一个datum联合：

```
union datum {
    int i;
    double d;
};
```

并且定义了一个该联合类型的变量：

```
union datum u;
```

为了在这个联合中存储一个整数，可以使用下面的写法：

```
u.i = 15;
```

为了在这个联合中存储一个浮点数，可以使用下面的写法：

```
u.d = 88.9e4;
```

只有当一个成员的类型与该联合的最后一次赋值的类型相同时，才能够引用这个成员。C并没有提供方法查询联合最后所赋值的是哪个成员，程序员可以记住它或以显式地记录与联合相关的数据标签。数据标签是个与联合相关联的对象，提示当前在联合中所存储的是哪个成员的值。数据标签和联合可以包含在一个公共的结构中。

例子

我们可以把下面这个联合

```
union widget { long count; double value; char name[10];} x;
```

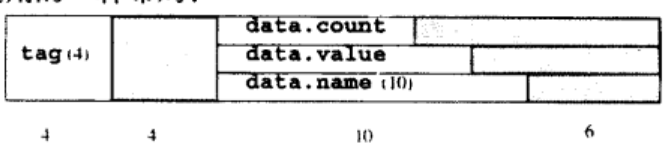
替换为

```
enum widget_tag { count_widget,
                  value_widget,
                  name_widget };

struct WIDGET {
  enum widget_tag tag;
  union { long count;
          double value;
          char name[10]; } data;
} x;

typedef struct WIDGET widget;
```

widget结构的长度是24个字节（它假设double类型的对象必须按照8字节的边界对齐）。下面显示了这个结构可能使用的一种布局：



一种常见的情况是，如果double类型的对象可以存储在4字节的边界上，则widget的长度只需要16字节。

为了把一个整数赋值给这个联合，可以采用下面的方法：

```
x.tag = count_widget;
x.data.count = 10000;
```

为了把一个浮点数赋值给这个联合，可以采用下面的方法：

```
x.tag = value_widget;
x.data.value = 3.1415926535897932384;
```

为了把一个字符串赋值给这个联合，可以工具函数strncpy：

```
x.tag = name_widget;
strncpy(x.data.name, "Millard", 10);
```

下面是一个可移植的函数，它可以区分这个联合的各种不同情况。在调用print_widget时，可以不管最后一个赋值的是哪个成员：

```
void print_widget(widget w)
{
    switch(w.tag) {
        case count_widget:
            printf("Count %ld\n", w.data.count); break;
        case value_widget:
            printf("Value %f\n", w.data.value); break;
        case name_widget:
            printf("Name \"%s\"\n", w.data.name); break;
    }
}
```

尽管标准C对联合的布局只作了很少的保证，但它对包含了一些类似结构类型的成员的联合作出了一个特殊的保证。如果这些结构都是从它们的结构的相同起始序列开始，标准C就保证这些初始序列能够正确地与其他序列重合。因此，可以在每个结构的前面放置一个数据标签，这样就可以使用任何结构成员来引用这个标签。

参考：类型转换表达式 第7.5.1节；枚举 第5.5节；重叠 第4.2.4节；作用域 第4.2.1节；switch语句 第8.7节；strncpy工具函数 第13.3节；结构 第5.6节；typedef 第5.10节。

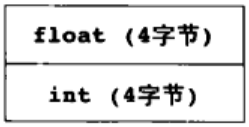
5.7.4 （其他）使用联合类型

如果对联合的最后一次赋值并不是通过同一个成员进行时，对联合成员的引用就是不可移植的。有时候，程序员会使用这种方法“深入”到C类型的底层，发现和计算机的底层数据表示形式有关的一些信息（它本身就是个不可移植的概念）。

例子

为了发现一个浮点数是如何表示的：

1. 创建一个联合，包含了相同长度的浮点类型和整型成员：



2. 对浮点成员进行赋值。

3. 读取整型成员的值，并把它打印出来（例如打印为十六进制数的形式）。

下面是一个完成这些任务的函数，它假设float类型和int类型具有相同的长度：

```
void print_rep(float f)
{
   union { float f; int i } f_or_i;
   f_or_i.f = f;
   printf("The representation of %12.7e is %#010x\n",
      f_or_i.f, f_or_i.i );
}
```

当调用print_rep(1.0)时，在基于Motorola 68020的工作站中，输出结果是：

```
The representation of 1.0000000e+00 is 0x003f800000
```

注意，无法使用类型转换操作符来发现底层表示形式。C的转换类型操作符把它的操作数转换为新表示形式中的最接近值。(int) 1.0是1，而不是0x003f80000。

5.8 函数类型

“返回T的函数”类型是一种函数类型，其中T可以是除了数组或函数之外的任何类型。换句话说，函数不能返回数组或函数，但可以返回指向数组或函数的指针。

函数只能通过两种方式引入到程序中。首先，函数定义可以创建一个函数，指定它的形参和返回值，并提供该函数的函数体。第9.1节提供了和函数定义有关的更多信息。其次，函数声明可以引入对一个在其他地方定义的函数的引用。

例子

下面是square函数的定义：

```
int square(int x)
{
   return x*x;
}
```

如果square是在其他地方定义的，下面这个声明就引入了它的名称，并允许调用这个函数。

```
extern int square(int);
```

外部函数声明可以引用在其他C源文件中声明的函数，也可以引用在同一个源文件的后面所定义的函数（即“前向引用”）。

例子

前向引用可以用于创建相互递归的函数，例如下面例子中的f和g：

```
extern int f(void);
...
int g(void) {...f();...}
int f(void) {...g();...}
```

相同的声明风格还可以用于静态函数：

```
static int f();
...
static int g() {...f();...}
static int f() {...g();...}
```

有些非标准的C编译器可能不允许对静态函数进行这种类型的前向引用。有时候它们会稍作妥协，允许第1个声明使用存储类别指定符extern，但在函数定义时把存储类别修改为static。

例子

```
extern int f(void);  /* not really extern, see below... */
...
static int g(void) {...f();...}
static int f(void) {...g();...} /* now, make f static */
```

这种编程惯用法至少容易产生误导。标准C要求函数（事实上，包括任何标识符）的第1个声明指定它是extern还是static。这就允许区别对待静态函数和外部函数的编译器在这种情况下只进行一道编译就可以了。标准C并没有显式地禁用“先extern然后static风格”，但并没有指定它的含义。

对函数类型表达式的唯一操作是把它转换为一个指向函数的指针并调用它。

例子

在下面这个声明中，外部标识符f、fp和apf的类型分别是“返回int的函数”、“指向返回int的函数的指针”和“指向接受1个double形参并返回int的函数的指针的数组”：

```
extern int f(), (*fp)(), (*apf[])(double);
```

apf的声明包含了一个标准C的函数原型。这些标识符可以通过下面的代码在函数调用表达式中使用。

```
int i,j,k;
...
i = f(14);
i = (*fp)(j, k);
i = (*apf[j])(k);
```

当一个没有可见原型的函数被调用时，实际参数会执行一些标准转换，但编译器并不会检查实参的类型或数量是否与函数已知的形式参数的类型或数量匹配。具有可见原型的函数的实参会转换为指定的形参类型。在前面这个例子中，*apf[j]所指定的整数实参k将被转换为double类型。

在标准C以及其他一些C编译器中，可以在函数调用中使用“指向函数的指针”类型的表达式，而不必进行显式的解引用。在这种情况下，前面这个例子所调用的(*fp)(j, k)可以改写为fp(j, k)。

“返回……的函数”类型的表达式如果不是出现在函数调用中、不是作为取地址操作符&或者sizeof操作符的操作数，将会立即转换为“指向返回……的函数的指针”类型（作为sizeof操作数的函数不会执行转换，这就保证这种sizeof表达式将是非法的，而不是产生指针的长度）。仅有的能够产生“返回T的函数”类型的值的表达式就是这个函数的名称以及由单目间接访问操作符*（作用于“指向返回……的函数的指针”类型的表达式）所组成的间接访问表达式。

例子

下面这个程序把同一个指针值赋值给fp1和fp2：

```
extern int f();
int (*fp1)(), (*fp2)();
```

```
fp1 = f;   /* implicit conversion to pointer */
fp2 = &f;  /* explicit manufacture of a pointer */
```

调用一个函数所需要的所有信息被封装在一个“指向返回……的函数的指针”类型的对象中，尽管函数指针常常被认为是这个函数的代码在内存中的地址。在有些计算机上，函数指针实际上指向调用这个函数所需要的一块信息。这种和表示形式有关的问题对于C程序员而言通常是不可见的，只有编译器的实现者需要关心这个问题。

参考：函数实参转换 第6.3.5节；函数调用 第7.4.3节；函数声明器 第4.5.4节；函数定义 第9.1节；函数原型 第9.2节；间接访问操作符* 第7.5.7节；sizeof操作符 第7.5.2节；寻常单目转换 第6.3.3节。

5.9 void 类 型

void类型不包含任何值或操作。

void类型指定符：

```
void
```

void类型用于

- 作为函数的返回类型，表示这个函数不返回任何值。
- 用于类型转换表达式中，显式表示丢弃一个值。
- 形成void *类型，它是一种“通用的”数据指针。
- 在函数声明器中代替形参列表，表示这个函数不接受任何参数。

例子

write_line的声明使用了void，它既作为返回类型，又作为形参列表。

```
extern void write_line(void);
…
write_line(); /* no value returned */
```

write_line2的声明表示这个函数返回一个值，但这个函数调用把它转换为void，显式地丢弃了这个返回值。

```
extern int write_line2(void);
…
(void) write_line2(…);    /* ignore returned value */
```

参考：类型转换 第7.5.1节；丢弃表达式 第7.13节；void * 第5.3.2节。

5.10 typedef 名 称

在编写一个声明时，如果它的“存储类别”是typedef，就会调用类型定义工具。

typedef名称：
 标识符

在这种声明的任何声明器中所封装的标识符被定义为一种类型的名称（即typedef名称）。如果这个声明是个常规的变量声明，这个类型就是由这个标识符所指定的。当一个名称被声明为一种类型时，它可以出现在类型指定符允许出现的地方。这就允许为复杂的类型提供容易记忆的简写形式。

例子

考虑下面这些声明：

```
typedef int *IP;       /* IP: "pointer to int" */
typedef int (*FP)( ); /* FP: "pointer to function
                         returning int" */
typedef int F(int);    /* F: "function with one int
                          parameter, returning int" */

typedef double A5[5]; /* A5: "5-element array of double" */
typedef int A[];      /* A: "array of int" */
```

在前面这些声明之后，就允许使用下面这些声明：

```
IP ip;       /* ip: pointer to an int */
IP fip();    /* fip: function returning a pointer to  int */
FP fp;       /* fp: pointer to a function returning int */
F *fp2;      /* fp2: pointer to a function taking an
                  int parameter and returning an int */

A5 a5;       /* a5: 5-element double array */
A5 a25[2];  /*  a25: double [2][5]: a 2-element array
                 of 5-element arrays of double */
A a;         /* a: array of int (with unspecified bounds) */
A *ap3[3];   /* ap3: 3-element array of pointers to
                 arrays of int (with unspecified bounds) */
```

例子

typedef名称不能与其他类型指定符进行组合：

```
typedef long int bigint;
unsigned bigint x;       /* invalid */
```

对typedef名称使用类型限定符是允许的，并且非常实用：

```
const bigint x;    /* OK */
```

使用typedef存储类别指定符的声明并不会引入新的类型，这个名称被认为是以其他方式所指定的另一个名称的同义词。

例子

在下面这个声明之后，

```
typedef struct S { int a; int b; } s1type, s2type;
```

类型指定符s1type、s2type和struct S可以互换地使用，它们表示同一种类型。

尽管typedef只是引入了以其他方式所命名的类型的同义词，但C编译器可能想在内部保留被声明的类型名称，使调试器和其他工具可以通过名称引用程序员所使用的类型。

在C99中，如果一个typedef声明包含了一个可变长度的数组类型，则这个可变长度表达式是当这个typedef声明被处理时进行求值的，而不是当这个typedef名称声明这个数组时进行求值的。

例子

在下面的代码片段中，数组a是个包含10个整数元素的数组，因为Array类型的长度在看到这个typedef声明时就已固定，而不是在声明a的时候。

```
{
   int n = 10;
```

```
    typedef int Array[n];
    n = 25;
    Array a;
    ...
}
```

参考：类型兼容 第5.11节；可变长度的数组 第5.4.5节。

5.10.1 用于函数类型的typedef名称

可以为函数类型提供一个typedef名称，但函数无法从typedef名称继承它们的“函数功能”。这多少对函数typedef名称产生了一些限制。

例子

通过下面这个声明，DblFunc成为“返回double的函数”的同义词：

```
typedef double DblFunc();
```

一旦声明之后，DblFunc可以根据声明器的常规组合规则，声明指向函数类型的指针、指向函数类型的指针数组等：

```
extern DblFunc *f_ptr, *f_array[];
```

除了遵守常规的类型声明规则之外，程序员不能声明非法的类型，例如函数数组：

```
extern DblFunc f_array[10];   /* Invalid! */
```

但是，DblFunc不能用于定义函数。下面这个fabs的定义将被拒绝，因为它看上去定义了一个返回另一个函数的函数。

```
DblFunc fabs(double x)
{
    if (x<0.0) return -x; else return x;
}
```

另外，不能通过省略fabs后面的括号来解决这个问题，因为参数列表必须出现在这对括号中。函数定义必须以通常的方式编写，就当DblFunc并不存在：

```
double fabs(double x)
{
    if (x<0.0) return -x; else return x;
}
```

在标准C中，typedef名称可以包含函数原型信息，包括形参名称：

```
typedef double DFuncType( double x );
typedef double (*FuncPtr)( int, float );
```

在这个例子中，DFuncType是一种函数类型，FuncPtr是一种函数指针类型。

参考：函数声明器 第4.5节；函数的定义 第9.1节；函数的原型 第9.2节。

5.10.2 重定义typedef名称

C语言指定了typedef名称可以按照和普通的标识符一样的风格在内层代码块中进行重定义。

例子

```
typedef int T;
T foo;
...
```

```
{
    float T; /* New definition for T */
    T = 1.0;
    ...
}
```

其中一个限制是在重定义中无法省略类型指定符，即使类型将默认为int。有些非ISO的编译器在typedef名称的重定义方面存在一些问题，这可能是由于typedef名称对C语言的语法所造成的压力。现在我们转到这个问题。

参考：在C++中重定义typedef名称 第15.13.2节；名称的作用域 第4.2.1节。

5.10.3 编译器的说明

允许普通标识符作为类型指定符使C的语法对上下文环境更加敏感，因此不符合LRLR（1）。为此，考虑下面的程序行

```
A ( *B ) ;
```

如果A被定义为一个typedef名称，则这行代码是把变量B声明为“指向A的指针”类型（*B两边的括号被忽略）。如果A并不是一个类型名称，则这行代码就是调用函数A，并以*B作为唯一的参数。这种歧义性无法通过语法来解决。

基于解析生成器YACC的C编译器（例如Portable C编译器）通过从语义分析阶段向词法分析阶段发送反馈信息来解决这个问题。所有的C编译器都必须在词法分析阶段处理typedef。

5.11 类型兼容性

在C中，如果两种类型属于同种类型，或者“足够接近”，在许多情况下可以看成是相同的，这两种类型就是兼容的。类型兼容的概念是由标准C引入的，但它只是以一种更正规的形式捕捉了传统C所使用的规则。标准C还需要其他一些规则，以处理标准C的一些特性，例如函数原型和类型限定符。对于两种兼容的类型，它们可以是相同的类型，也可以是具有某些共同属性的指针、函数或数组。后面的小节将讨论这些特殊的规则。

每两个兼容类型具有一个相关联的合成类型（composite type），它是由两种兼容类型所产生的公共类型。它有点类似于使用寻常双目转换对两个整数类型进行组合，产生一个公共的结果类型，以便在一些算术操作符中使用。我们将在描述类型兼容性规则时描述由两个兼容类型所产生的合成类型。

参考：数组类型 第5.4节；函数原型 第9.2节；函数类型 第5.8节；指针类型 第5.3节；结构类型 第5.6节；类型限定符 第4.4.3节；联合类型 第5.7节；常用二进制转换第6.3节。

5.11.1 等价的类型

只有当两种算术类型是同一种类型时，它们才是兼容的。如果一种类型可以用其他不同的类型指定符组合来表示，这些组合形式都属于同一种类型。例如，short和short int是相同的，但unsigned、int和short是各不相同的类型。signed int类型与int类型是相同的（除了在表示位段的类型时）。short和long与它们对应的singed版本也存在这个关系。char、signed char和unsigned char总是不同的。

如果两种类型是相同的，它们必然是兼容的，并且它们的合成类型也属于同一种类型。在标准C中，任何限定符都可能会改变类型。例如，const int和int并不兼容，当然更不是相同的类型了。通过typedef为一种类型所声明的名称总是该类型的同义词，而不是一种新的类型。

例子

在这些声明之后，p和q的类型是相同的。x和y的类型是相同的，但它们都和u的类型不同。类型TS和struct S是相同的，u、w和y的类型也是相同的。

```
char * p, *q;
struct {int a, b;} x, y;
struct S {int a, b;} u;
typedef struct S TS;
struct S w;
TS y;
```

例子

在这些声明之后，my_int的类型与int相同，my_function的类型和“float *()”相同：

```
typedef int my_int;
typedef float *my_function();
```

例子

在这些声明之后，变量w、x、y和z都具有相同的类型。

```
struct S { int a, b; } x;
typedef struct S t1, t2;
struct S w;
t1 y;
t2 z;
```

参考：整数类型 第5.1节；指针类型 第5.3节；结构类型 第5.6节；typedef名称 第5.10节。

5.11.2 枚举兼容性

每个枚举类型定义产生了一种新的整数类型。标准C要求每种枚举类型与编译器所定义的表示它的整数类型保持兼容。对于同一个程序中的不同枚举类型，它们的兼容整数类型可能不同。合成类型是枚举类型。同一个源文件所定义的两个不同的枚举类型是不兼容的。

例子

在下面的声明中，E1和E2的类型是不兼容的。但E1和E3的类型是兼容的，因为它们属于同种类型。

```
enum e {a,b} E1;
enum {c,d} E2;
enum e E3;
```

由于枚举类型一般被当做整数类型，因此不同枚举类型的值可以混合使用，而不必考虑类型兼容性。

例子

兼容性规则的效果是标准C将拒绝下面的第2个函数声明，因为原型中的参数类型与第1个声明并不一致：

```
extern int f( enum {a,b} x);
extern int f( enum {b,c} x);
```

非标准编译器有时会把枚举类型看成与int类型完全兼容，并且各种枚举类型之间也完全兼容。

参考：枚举类型 第5.5节。

5.11.3 数组兼容性

对于两个具有相似限定形式的数组，只有当它们的元素类型兼容时，它们才是兼容的。如果这两种数组类型都指定了常量长度，则它们的长度必须相等。但是，如果只有一个数组类型指定了常量长度，或者两者都未指定常量长度，则这两种类型也是兼容的。两个兼容数组类型的合成类型就是具有合成元素类型的数组类型，并且具有和那两个数组类型相同的类型限定形式。如果任一源类型指定了一个常量长度，则合成类型也是这个常量长度。否则，它的长度就是未指定的。如果数组的长度并未指定，并且使用它们的上下文环境要求它们兼容，则它们在运行时的长度必须相同，否则结果就是未定义的。

例子

下面这些数组的兼容情况如注释所示。e是一个可变长度的数组（C99）。

```
extern int a[];   /* compatible with b, c, and e; not d */
int b[5]; /* compatible with a          and e only */
int c[10]; /* compatible with a         and e only */
const int d[10]; /* not compatible with other types */
int e[n]; /* compatible with a, b, and c; not d */
```

d的类型与其他类型不兼容，因为它的元素类型const int与元素类型int不兼容。类型a和b的合成类型是int [5]。在运行时，只有当a的实际定义长度为5时，a和b才能够互换地使用。

参考：数组类型 第5.4节；数组声明器 第4.5.3节；类型限定符 第4.4.3节；可变长度数组 第5.4.5节。

5.11.4 函数兼容性

为了使两个函数类型保持兼容，它们必须指定兼容的返回类型。如果两个函数类型都是按照传统形式（非原型）指定的，就没有其他要求了。它们的合成类型就是具有合成返回类型的函数类型（传统形式）。

为了使两个采用原型形式声明的函数保持兼容，必须满足下面这些条件：

1. 函数的返回类型必须是兼容的。
2. 形参的数量以及省略号的使用必须一致。
3. 对应的形参必须具有兼容的类型。

形参的名称并不需要一致。合成类型是满足形参为合成形参类型、以相同的方式使用省略号并且具有兼容返回类型的函数类型。

如果两个函数类型中只有一个采用了原型形式，为了使这两个函数类型保持兼容，必须满足下面这些条件：

1. 返回类型必须兼容。
2. 原型不能包含省略号终止符。
3. 原型中的每个形参类型T必须与对T进行寻常实参转换时所产生的类型兼容。

它们的合成类型是具有合成返回类型的原型形式的函数类型。

参考：函数原型 第9.2节；函数类型 第5.8节。

5.11.5 结构和联合兼容性

在每个结构类型定义或联合类型定义中，它的类型标识符引入了一种新的结构或联合类型。它们并不与同一个源文件中所定义的其他结构或联合相同，也不与它们兼容。

当一个类型指定符是结构、联合或枚举类型引用时，它与对应的类型定义所引入的类型相同。类型标签用于把引用与定义相关联。从这层意思上说，它们都是类型的名称。

例子

下面的x、y和u的类型都是不同的，但u和v的类型是相同的：

```
struct { int a; int b; } x;
struct { int a; int b; } y;
struct S { int a; int b; } u;
struct S v;
```

参考：枚举 第5.5节；结构 第5.6节；联合 第5.7节。

5.11.6 指针兼容性

对于两个具有相似限定形式的指针，当它们指向兼容的类型时，它们就是兼容的。两个兼容的指针类型的合成类型是指向合成类型的指针（采用相似的限定形式）。

5.11.7 源文件之间的兼容性

尽管结构、联合和枚举类型的定义产生了新的（非兼容）类型，但还是需要有一种办法，允许同一个程序中单独编译的源文件之间的引用。

例子

假设一个头文件包含了下面这些声明：

```
struct S {int a,b;};
extern struct S x;
```

当一个程序中的两个源文件都包含了这个头文件时，其用意是这两个文件引用了同一个struct S类型的变量x。但是，从理论上说，每个文件包含了一个不同的结构类型的定义，只是它们恰好都命名为struct S。

除非同种类型的两个声明是兼容的，否则标准C规定程序的运行时行为是未定义的，因此：

1. 对于在不同的源文件中定义的两个结构或联合，只有当它们具有相同的成员，并且这些成员按照相同的顺序出现，并且对应成员具有兼容类型（包括位段的宽度）时，它们才是兼容的。在C99中，这个规则更进一步，要求结构标签也相同（或者都被省略）。

2. 对于在不同的源文件中定义的两个枚举，只有当它们包含了相同的枚举常量（可以是任何顺序），并且每个对应的枚举常量具有相同的值时，它们才是兼容的。

在这些情况下，合成类型就是当前源文件中的类型。

参考：枚举类型 第5.5节；结构类型 第5.6节；联合类型 第5.7节。

5.12 类型名称和抽象声明器

在C编程中，在两种情况下只需要编写类型的名称而不必声明这种类型的对象：当编写类型

转换表达式以及对一种类型应用sizeof操作符的时候。在这两种情况下，使用了一种通过抽象声明器创建的类型名称（不要把类型名称与“typedef名称”搞混）。

类型名称：

声明指定符 抽象声明器$_{opt}$

抽象声明器：

指针
指针$_{opt}$ 直接抽象声明器

指针：

* 类型限定符列表$_{opt}$
* 类型限定符列表$_{opt}$ 指针

类型限定符列表： (C89)

类型限定符
类型限定符列表 类型限定符

直接抽象声明器：

(抽象声明器)
直接抽象声明器$_{opt}$ [常量表达式$_{opt}$]
直接抽象声明器$_{opt}$ [表达式] (C99)
直接抽象声明器$_{opt}$ [*] (C99)
直接抽象声明器$_{opt}$ [形参类型列表$_{opt}$]

抽象声明器与常规的声明器相似，区别在于后者所封装的标识符被空字符串所代替。因此，类型名称看上去就像是省略了被封装的标识符的声明。根据语法，声明指定符不能包含存储类别指定符。形参类型列表只在标准C中得到支持，它用于原型形式的类型声明。

抽象声明器的优先级与常规的声明器相同。

例子

类型名称	翻译
int	int类型
float *	指向float的指针
char (*)(int)	指向接受一个int参数并返回char的函数的指针
unsigned *[4]	包含4个指向unsigned的指针的数组
int (*(*)())()	指向返回指向返回int的函数的指针的指针

类型名称总是出现在括号内，形成了类型转换或sizeof操作符语法的一部分。如果类型名称中的类型指定符是结构、联合或枚举类型定义，标准C要求编译器在这个位置用这个类型指定符所包含的类型标签（如果有）定义一个新类型。这个特性被认为是一种不良的编程风格（它在C++中是非法的）。

例子

假设在遇到下面这两条语句时，struct S还没有被定义（好的C编译器应该对第1行代码产生一条警告信息）。

```
i = sizeof( struct S {int a,b;}); /* OK, but strange */
j = sizeof( struct S ); /* OK, struct S is now defined */
```

参考：类型转换 第7.5.1节；函数原型 第9.2节；sizeof操作符 第7.5.2节。

5.13 C++兼容性

5.13.1 枚举类型

一种良好的编程风程是不要在不使用显式类型转换的情况下把枚举类型或枚举常量作为整数类型使用。和C不同，C++会区分不同的枚举类型，并且区分枚举类型和整数类型，尽管可以通过类型转换在它们之间转换。C++也允许从枚举类型到整数类型的隐式转换。

例子

```
enum e {blue, red, yellow} e_var;
int i_var;
…
i_var = red;          /* valid in both C and C++ */
e_var = 1; /* valid in C, not in C++ */
i_var = (int) red; /* valid in both C and C++ */
e_var = (enum e) 3;   /* valid in both C and C++ */
assert(sizeof(blue) == sizeof(int));
   /* always succeeds in C; may fail in C++ */
```

参考：枚举类型 第5.5节。

5.13.2 typedef名称

在C一样，typedef名称可以在内层作用域中被重新声明。但是，在C++中，对于结构和联合则不允许这种做法，因为它们具有自己的作用域，就像原来那个typedef名称已经在结构或联合中使用一样。这种情况实际上不太可能发生。

例子

```
typedef int INT;
struct S {
   INT i;
   double INT; /* OK in C, not C++; everywhere a bad idea*/
}
```

参考：重定义typedef名称 第5.10.2节。

5.13.3 类型兼容性

C++并不具有C的类型兼容性的概念。为了实行更严格的类型检查，C++在C只要求兼容类型的场合要求使用相同的类型。在有些情况下，如果类型不同，C++将会产生诊断信息。但是，由于C++提供了对C的“基本兼容性”，所以即使C++程序包含了未检测到的类型不相同但兼容（标准C）的情况，它仍然能够正确地工作。

参考：类型兼容性 第5.11节。

5.14 练 习

1. 为了表示下面各组值，应该选择什么C类型？假设最重要的目标是保证不同编译器和计算机之

间的移植性，其次是尽量减少空间消耗。

(a) 5位数字的美国邮政编码。

(b) 由3位区号和7位本地号码所组成的电话号码。

(c) 值0和1。

(d) 值−1、0和1。

(e) 字母值或−1这个值。

(f) 银行账户的余额，用美元和美分表示，最高为9 999 999.99。

2. 有些流行的计算机支持一种扩展字符集，除了包含常规的ASCII字符之外，还包含了其他字符，这些字符的值的范围从128～255。假设char类型用8位表示。下面这个is_up_arrow函数在输入字符表示向上箭头时返回“真”，否则就返回“假”。假设UP_ARROW_KEY在目标计算机上具有适当的值，这个函数是否可以在不同的标准C编译器之间移植？如果不能，请改写这个函数，以实现这个目标。

```
#define UP_ARROW_KEY  0x86
...
int is_up_arrow(char c)
{
    return c == UP_ARROW_KEY;
}
```

3. 如果vp的类型是void *，cp的类型是char *，下面哪些赋值语句在标准C中是合法的？

(a) `vp = cp;`　　(c) `*vp = *cp;`

(b) `cp = vp;`　　(d) `*cp = *vp;`

4. 如果iv的类型是int [3]，im的类型是int [4][5]，重写下面这些表达式，不使用下标操作符。

(a) `iv[i]`

(b) `im[i][j]`

5. 下面这个函数f所返回的是哪个整数值？return语句中向int的类型转换是否必要？

```
enum birds {wren, robin=12, blue_jay};
int f()
{
    return (int) blue_jay;
}
```

6. 下面是一个结构类型以及一个该类型的变量的定义。编写一系列的语句，向这个结构的每个成员赋一个合法的值。如果这个结构的两个成员重叠，只向其中一个重叠的成员赋值。

```
struct S {
    int i;
    struct T {
        unsigned s: 1;
        unsigned e: 7;
        unsigned m: 24;
    } F;
    union U {
        double d;
        char a[6];
        int * p;
    } U;
} x;
```

7. 画出前一个问题所定义的结构的草图，使用第5.6.5节和第5.7.3节所使用的格式。假设底层计算机使用了字节编码，用8位表示char类型，用32位表示指针类型和int类型，用64位表示double类型。在第1张草图中，假设计算机采用大端字节顺序，并且位段从右向左包装在一个32位的字中。在第2张草图中，假设计算机采用小端字节顺序，位段也是从右到左包装在字中。假设编译器在两种情况尽量紧密地包装位段（字节顺序在第6.1.2节讨论）。
8. 编写“返回指向整数的指针的函数”类型的typedef定义。编写一个变量的声明，它包含了一个指向这类函数的指针，并实际编写一个这样的函数，尽可能使用typedef定义。
9. 编写一个stdbool.h头文件，使程序员可以在C89编译器中使用C99风格的布尔类型。是不是存在什么限制？
10. 重新编写第5.7.3节的数据标签例子，包括print_widget函数。使WIDGET成为3个结构的联合，每个结构包含一个数据标签和一个数据值。你的实现是否可以移植到遵循标准C的编译器？



第6章 转换和表示形式

大多数编程语言试图隐藏语言在特定计算机上的实现细节。绝大多数C程序员而言并不需要关心这些细节，尽管C的一个主要吸引力是，允许程序员深入到抽象语言层次的下面探索程序和数据的底层表现形式。这种自由带来了一定的风险：有些C程序员可能会无意地深入到抽象层的下面，在程序中引入和数据表示形式有关的假设，而这种假设往往是不可移植的。

本章共有三个目标。首先，讨论了数据和程序表示形式的一些特征，说明了表示形式的选择是如何影响C程序的。其次，详细讨论了数据值从一种类型到另一种类型的转换，强调了C语言在不同的计算机上可以移植的一些特征。最后，讨论了C的“寻常转换规则”，它们是在表达式进行求值时自动执行的。

6.1 表示形式

本节讨论函数和数据的表示形式，并讨论表示形式的选择是如何影响C程序以及C编译器的。

6.1.1 存储单位和数据长度

在程序运行时，C的所有数据对象（除了位段之外）在计算机内存中是由整数数量的抽象存储单位表示的。每个存储单位又是由固定数量的“位”组成，每个位可以表示两个值之一，即0和1。每个存储单位必须可以唯一地编址，并且与char类型具有相同的长度。在C中，存储单位的位数因编译器而异，但它必须能够容纳基本字符集中的每个字符。标准C又把存储单位称为字节，但字节这个术语通常理解为正好由8个位所组成的存储单位。

根据定义，数据对象的长度就是该数据对象所占据的存储单位的数量。1个存储单位就是1个字符所占据的存储空间的数量。因此，char类型的对象的长度是1。字符的位数（字节）是由limits.h所定义的CHAR_BIT的值所指定的。

由于一种特定类型的所有数据对象占据相同的存储空间，我们也可以把类型的长度称为该类型的对象所占据的存储单位的数量。sizeof操作符可以确定一个数据对象或类型的长度。如果一种类型的长度大于另一种类型，我们可以认为前者“长于”或“大于”后者。类似，如果一种类型的长度小于另一种类型，我们可以认为前者“短于”或“小于”后者。标准C要求整数类型和浮点类型具有确定的最小范围，并提供了因编译器而异的头文件limits.h和float.h，定义了它们的长度。

例子

下面这个C99程序判断C的主要数据类型的长度。为了与旧版本的C兼容，%3zd 中的长度限定符z应该被一个适合size_t（sizeof的类型）的限定字符所代替：如果该类型是long，这个限定字符就是l。如果该类型是int，这个限定字符就是空。

```
#include <stdio.h>
int main(void)
{
    printf("\tType sizes:\n");
    printf("char\tshort\tint\tlong\tllong\t"
           "float\tdouble\tldouble\n");
    printf("%3zd\t%3zd\t%3zd\t%3zd\t%3zd\t"
           "%3zd\t%3zd\t%3zd\n",
           sizeof(char), sizeof(short), sizeof(int),
           sizeof(long), sizeof(long long),
           sizeof(float), sizeof(double),
           sizeof(long double));
    return 0;
}
```

参考：字符类型 第5.1.3节；float.h 第5.2节；limits.h 第5.1.1节；最小整数长度 第5.1.1节；sizeof操作符 第7.5.2节；stdio.h标准I/O头文件 第15章。

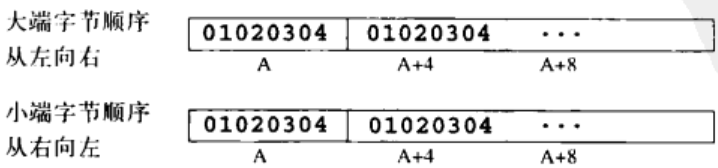
6.1.2 字节顺序

计算机的寻址结构决定了指针是如何对各种长度的存储片段进行引用的。对C而言，最自然的地址模型就是计算机内存中的每个字符（字节）都可以单独编址。使用这种模型的计算机称为字节编址的计算机。例如，一大片用于存储整数或浮点数的存储空间的地址一般与这片地址的第1个字符的地址相同。“第1个”字符就是具有最低地址的字符。

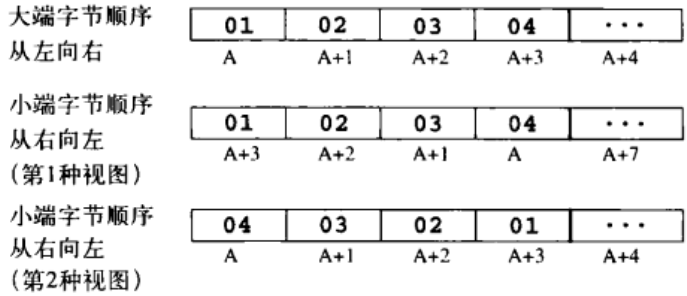
即使是在这种简单的模型中，不同的计算机在存储的“字节顺序”方面存在不同。也就说，它们在认为存储空间的哪个存储字节是第1个字节方面存在区别。在“从右向左”或“小端字节顺序”的体系结构（包括Intel 80x86和Pentium微处理器）中，32位整数的地址也是这个整数的低端字节的地址。在“从左向右”或“大端字节顺序”的体系结构（包括Motorola 680x0微处理器家族），32位整数的地址就是这个整数的高端字节的地址。有些嵌入式处理器可以配置为根据整体系统的需要采用大端或小端字节顺序。

例子

Intel（小端）和Motorola（大端）体系结构都是字节编址的，使用8位的字节和4字节的字，每个字可以存储一个32位的整数。下面这张图显示了在每个体系结构下一系列的字，每个字包含了32位的值0x01020304。你可以看到，这两种体系结构在这一层次上看上去是相同的。



当我们观察一个字内部的每个字节的内容时，情况就发生了变化。在大端字节顺序中，字的地址就是最左边那个字节（高端）的地址。由于字节地址从左向右递增，因此它看上去与我们以前绘制字的方法相同。但是，在小端字节顺序中，字的地址是最右边那个字节（低端）的地址。 我们可以按照两种方式想象这种结构：一种是把字的地址看成是从右向左递增，另一种是把字节看成是反向的。下面显示了这两种视图。



结构类型的成员是按照递增的地址分配内存的，也就是说，它是从左向右还是从右向左取决于计算机的字节顺序。由于位段也是根据字节顺序进行包装的，因此按照同样的约定对存储空间的位进行编号也是极为自然的。因此，在一台从左向右的计算机中，32位整数的最高位的位编号是0，最低位的位编号是31。在一台从右向左的计算机中，最低位的位编号是0，最高位的位编号是31。如果程序依赖一种特定的字节顺序，它就是不可移植的。

例子

下面这个程序以一种不可移植的方式使用一个联合确定计算机的字节顺序。这个联合与long类型的对象具有相同的长度，并进行了初始化，使这个联合的低端字节包含1，其他所有字节包含0。在从右向左的体系结构中，这个联合的字符成员Char将与long类型的成员Long的低端字节重合。在从左向右的体系结构中，这个联合的字符成员Char将与Long成员的高端字节重合：

```
#include <stdio.h>
union {
    long Long;
    char Char[sizeof(long)];
} u;

int main(void)
{
    u.Long = 1;
    if (u.Char[0] == 1)
        printf("Addressing is right-to-left\n");
    else if (u.Char[sizeof(long)-1] == 1)
        printf("Addressing is left-to-right\n");
    else printf("Addressing is strange\n");
    return 0;
}
```

6.1.3 对齐限制

有些计算机允许数据对象位于内存中的任何地址，而不管该数据的类型是什么。其他计算机则对有些数据类型施加了对齐限制，要求这些类型的对象只能位于某些地址。例如，采用字节编码的计算机要求32位（4个字节）整数的地址是4的倍数是一种极为常见的限制。在这种情况下，我们认为这些整数的“对齐模数”是4。不遵守对齐限制可能会导致运行时错误或意想不到的程序行为。即使没有对齐限制，使用未对齐的地址也可能导致性能上的损失。因此，C编译器可能纯粹出于效率的原因要求数据对齐。

正常情况下，C程序员并不会意识到对齐限制，因为编译器会负责把数据放在适当的地址边界上。但是，C通过把指针转换为不同的类型，向程序员提供了违背对齐限制的能力。未初始化

的指针也可能违反对齐限制。

一般而言，如果类型S的对齐要求至少和类型D一样严格（对S的对齐模数不小于D的对齐模数），则把“指向类型S的指针”转换为“指向类型D的指针”是安全的。这里所说的“安全”是指转换所产生的指向类型D的指针如果用于存储或读取D类型的对象将会表现出合乎预期的行为，并且以后再转换回原先的指针类型将恢复为原来的指针。这个准则的一个必然推论是任何数据指针都可以安全地转换为char *或void *类型的指针，并可以转换回来。因为这两种指针类型的对齐要求是最低的。

如果类型S的对齐要求不如类型D严格，则把“指向类型S的指针”转换为“指向类型D的指针”可能导致两种预想不到的行为之一。首先，试图使用转换所产生的指针读取或存储D类型的对象可能会出现错误，使程序终止。其次，硬件或编译器可能会“调整”目标指针，使它成为合法的指针，通常是把它转换到前一个最接近的合法地址。以后再转换回到原先的指针类型可能无法恢复为原指针。

参考：字节顺序 第6.1.2节；malloc函数 第16.1节；指针类型 第5.3节。

6.1.4 指针的长度

C并没有要求任何整数类型的长度必须能够表示指针，虽然C程序员经常假设long类型足以容纳指针的长度，并且在绝大多数计算机上也确实如此。在C99中，inttypes.h头文件可能定义了整数类型intptr_t和uintptr_t，它们保证足够大，可以以整数的形式保存指数。

虽然函数指针通常不会比void *指针更大，但它并不保证如此（如第6.1.5节所述）。标准C把对象和函数指针之间的任何转换都视作未定义的行为。

参考：函数类型 第5.8节；指针转换 第6.2.7节。指针类型 第5.3节；类型的长度 第6.1.1节。

6.1.5 地址模型的效果

本节描述了计算机的内存设计对程序员以及编译器实现者在某些方面所产生的影响。

内存模型 有些小型和特殊用途的微处理器在设计指针的表示形式时要兼顾时间和空间的平衡，不同的程序可能适合不同的表示形式。这些处理器可以同时利用“短”地址和“长”地址。较小的地址（位于单个段中）具有更高的效率，但限制了可以引用的内存数量。大型程序常常需要访问多个段。

为了满足不同程序的需要，这些计算机的C编译器常常允许程序员指定内存模型，后者确定了程序所使用时间—空间权衡。表6-1显示了早期PC的C编译器所支持的描述性内存模型。这些模型的一些变型至今仍然可以在一些数字信号处理器中找到。

这里需要注意几个要点。在所有的内存模型中，代码和数据位于单独的内存段中，它们具有各自的地址空间。因此，数据和函数指针包含相同的值是完全可能的，尽管前者指向对象，后者指向函数。在compact和medium内存模型中，数据和函数指针具有不同的长度。在使用null指针常量NULL（第5.3.2节）时需要细心，因为它是对象指针。在涉及函数指针的表达式中简单地使用NULL指针，将会导致它被适当地转换。但是，在没有函数原型的情况下，把NULL作为函数指针参数进行传递可能存在问题。在标准C中，这个问题可以通过使用函数原型来解决，它可以使实参得到正确的转换。

表6-1 早期PC上的内存模型

内存模型名称	数据指针长度	函数指针长度	特 征
tiny	16位	16	数据、代码和堆栈都占据一个单一的段
small	16	16	代码占据一个64KB的段，数据和堆栈占据另一个64KB的段
medium	16	32	代码可以占据多个段，数据和堆栈限制在一个段中
compact	32	16	代码和堆栈各自限制在一个64KB的段中，其他数据可以占据多个段
large	32	32	代码和数据都可以占据多个段，堆栈限制在一个段中
huge（32位平台）	32	32	和large相同，但单个数据项的长度可以超过64KB

例子

不熟悉段式体系结构的C程序员可能会以为数据指针和函数指针只有在它们都是null指针时才可能包含相同的值，因此可能会不正确地使用下面的测试。这个测试并不可行，因为cp和fp可能指向不同的地址空间，偶尔可能拥有相同的非null值。

```
char *cp;
int (*fp)();
...
/* See if cp and fp are both null */
if ((int)cp == (int)fp) ... /* Incorrect!! */
```

例子

下面这个例子来自传统C，当使用compact或medium内存模型时，函数f的行为是未定义的，因为作为实参传递的null指针是个数据指针，而不是函数指针，因此具有不正确的长度：

```
extern int f(); /* no parameter information */
...
f(NULL);    /* This is NOT OK! */
...
int f( int (*fp)() ) {...}
```

指针长度的显式控制 除了在整个程序中使用一种特定的内存模型之外，另一种方法是指定特定函数或数据对象所使用的“near”或“far”指针。按照这种方式，程序员可以避免由于范围过大而导致的性能下降。不过，采用这种做法，将会降低程序的移植性，并使程序变得不易维护。

例子

有些段式体系结构的C编译器定义了新关键字__near和__far，可以在变量和指针的声明中使用。从语法上说，它们可以出现在标准C的类型限定符可以出现的地方。这两个关键字以两个下划线开始，因为这类名字是编译器所保留的（第10.1.1节）。

```
char __near near_char, *cp;
int __far (*fp)(), big_array[30000]
```

它的意图是far指针将占据32位，而near指针则使用16位。声明为far的函数或数据对象可以被编译器放在远端的段中。但是，near函数或数据必须放在“根”段中。在使用这些语言扩展时，当程序员把指针传递给并不使用原型声明的函数时，必须非常小心。

数组地址 不管计算机是否使用段地址方案，有些计算机采用了这样的设计：当数组较小时（一般不超过64KB），访问数组元素具有很高的效率。为了使用较大的数组，程序员必须提供一

个特殊的编译器选项，或者以某种方式指定一个很大的数组。

非常困难的计算机 尽管C在许多计算机上具有非常高效的实现，但仍有一些计算机的数据和地址表示形式对于C编译器而言是非常笨拙的。当计算机的自然字长不是它的自然字节长度的倍数时，就会出现一个主要的问题。假设（这是一个真实的例子）计算机使用36位的字，并且字符用7位表示。每个字可以包含5个字符，并留有1个未使用的位。所有的非字符数据类型占据1个或多个完整的字。这种内存结构对于C编译器而言是非常难以实现的，因为C的编程依赖于能够把任何数据结构映射到字符数组。也就是说，把一个T类型的对象复制到地址A，它应该足以把sizeof(T)个字符复制到A开始的地址。在这种计算机上，编译器实现者可以采用的唯一方法是使用非标准的位数来表示字符（例如9或36），使它们可以紧密地放在字中。这种表示形式可能导致显著的性能降低。

在“字编址”的计算机中，如果基本的可编址存储单位大于单个字符，也会出现类似的问题。在这类计算机中，可能存在（也可能不存在）一类特殊的地址，称为“字节指针”，可以表示字中的字符。假设有一个这样的字节指针，它可能比一个指向非字符类型的对象的指针更大，或者在指针中使用了某些在其他类型的提针中将被忽略的位（在正常情况下被设置为0）。C编译器必须决定是否付出额外的开销，把所有的指针用字节指针来表示，或者只对char *类型（以及标准C中的void *类型）的对象使用较大的格式，或者使用全字来表示每个字符。如果使用不同长度的字符指针，C程序员在进行指针转换时必须格外小心。

参考：*数组类型* 第5.4节；*字符类型* 第5.1.3节；*函数实参转换* 第6.3.5节；*函数原型* 第9.2节；*指针类型* 第5.3节；*存储单位* 第6.1.1节。

6.1.6 类型表示形式

某些类型的值的表示形式就是这种类型的对象在内存区域中的位模式。这种模式把这个对象的值与该类型的其他可能的值区分开来。这种类型的表示形式并不一定要使用该类型对象中所有的位，有些位只是“填充位”，它们的值是未定义的。例如，short数据类型可能只使用16位，但存储在32位的字中。sizeof所返回的长度包括了填充位。如果忽略所有的填充位，则“范围”或“精度”是更准确的术语。

另一种可能出现的情况是一种类型的同一个值可能具有多种表示形式。例如，整数0存在+0和−0的表示形式。编译器可以自由地在任何时候选择其中一种表示形式。

一种类型的表示形式可能与其他类型的表示形式不兼容，即使它们具有相同的长度。如果在访问一个long类型的值时把它看成是float类型的，其结果是未定义的，甚至可能导致程序终止。

根据C99的术语，对象的*有效类型*（effective type）就是对象当前所使用的表示形式的类型。正常情况下，数据对象（例如变量）被声明为某种类型，它的有效类型就是这种类型，因此不存在问题。有时候，当一个对象是通过malloc所分配的时候，它可能并没有被声明为某种类型。这时候，这个对象的有效类型就是最后一次在这个对象中存储值时所使用的左值表达式的类型。以后访问这个对象时，必须通过与有效类型（或兼容类型的限定版本）兼容的类型，否则结果就是未定义的。如果把一个值复制给一个未声明类型的对象（例如使用memcpy或者引用存储对象的底层char值），将会导致原值的有效类型被目标对象所采纳。

参考：*左值* 第7.1节；malloc 第16.1节；memcpy 第14.3节；*限定类型* 第4.4节。

6.2 转 换

C语言允许在几种情况下把一种类型的值转换为另一种类型：

- 类型转换表达式可以把一个值显式地转换为另一种类型。
- 为了执行算术或逻辑运算，操作数的值可以隐式地转换为其他类型。
- 一种类型的对象可能会赋值给另一种类型的地址单元（左值），这会引起隐式类型转换。
- 函数的实际参数在函数调用之前可能会隐式地转换为另一种类型。
- 函数的返回值可能会在函数返回之前隐式地转换为另一种类型。

一个特定的类型并不能够转换为任何类型，而是存在一定的限制。例如，在赋值时可能出现的所有转换与强制类型转换时可能发生的所有转换并不相同。

在下面各节中，我们讨论程序中可能出现的各种转换，并讨论在前面所列出的各种情况下，这些转换实际上是如何执行的。

6.2.1 表示形式的变化

当一个值从一种类型转换为另一种类型时，可能会发生表示形式的变化，也可能不发生变化。例如，当两个类型具有不同的长度时，就必定出现表示形式的改变。当整数转换为浮点表示形式时，即使整数和浮点数具有相同的长度，也必定会出现表示形式的变化。但是，当一个int类型的值转换为unsigned int类型时，就并不一定会发生表示形式的变化。

一些表示形式的变化是很简单的，只是抛弃多余的位或者用0填充相应的位。其他改变可以是复杂的，例如，在整数和浮点数表示之间的转换。对于下节中讨论的每一种转换，我们都描述了可能需要的表示变化。

6.2.2 细微的转换

把一个值从一种类型转换为另一种相同（或兼容）的类型总是可行的。参见第5.11节对相同类型和兼容类型的讨论。在这种情况下，不会发生表示形式的改变。

大多数编译器拒绝把结构或联合类型转换为自身类型，因为一般不允许对结构或联合进行转换。

6.2.3 转换为整数类型

标量类型（算术类型和指针）可以转换为整数类型。

布尔转换 在C99中，涉及_Bool类型的转换与涉及其他整数类型的转换稍有不同。把算术类型转换为_Bool类型时，如果原值为0，转换后的值为0。否则，转换后的值就是1。把指针类型转换为_Bool类型时，null指针被转换为0，其他所有指针值被转换为1。把_Bool类型转换为算术类型时，其结果是转换到目标类型的0或1。在本节的剩余部分，除非另有所述，否则我们所提到的整数类型不包括_Bool类型。

整数类型转换为整数类型 除了_Bool类型之外，从一种整数类型转换为另一种整数类型的基本规则是转换结果的数学值应该尽可能等于原先的数学值。例如，如果一个无符号整数的值为15，把这个值转换为一种有符号整数类型时，所产生的值也应该是15。

如果无法用新类型表示一个对象的原值，则存在两种情况。如果结果类型是有符号类型，

则转换结果被认为是溢出，结果值从技术上说是未定义的。如果结果类型是无符号类型，则转换结果必须是结果类型的一个独一无二的值，并且等于同余模数2^n减去原值，其中n是表示结果类型所使用的位数。如果有符号类型是用补码记法表示的，则相同长度的有符号类型和无符号类型进行转换时，不需要表示形式的改变。但是，如果有符号类型是用其他记法表示的，例如反码或符号绝对值表示形式，则需要更改表示形式。

当一个无符号整数转换为一个相同长度的有符号整数时，如果原值太大，无法用有符号表示形式来表示，转换结果就被认为是溢出（即无符号数的高端位为1）。但是，许多程序员以及C程序依赖于静悄悄地执行这种转换，在不改变表示形式的情况下产生负数。

如果目标类型的长度大于原类型，则结果类型无法表示原值的唯一情况就是当负的有符号值转换为一种更长的无符号类型的时候。在这种情况下，转换行为必须就像首先把原值转换为与目标类型相同长度的有符号类型，然后再转换为目标类型一样。

例子

由于常量表达式−1的类型是int：

```
((unsigned long) -1) == ((unsigned long) ((long) -1))
```

如果目标类型的长度小于原类型，并且原类型和目标类型都是无符号类型，则转换结果就是简单地截去原值的高位。结果表示形式的位模式将等于原表示形式的n个低位，其中n是目标类型的位数。把采用补码表示形式的有符号整数转换为更短的无符号类型时，所使用的丢弃规则与前面相同。这个丢弃规则也是把有符号或无符号整数转换为更短的有符号整数（采用补码表示形式）可以接受的几种方法之一。注意，这个规则在发生溢出时并不会保留符号位。不过，溢出后的行为在任何情况下都没有定义。当有符号整数并不是用补码表示时，转换就更为复杂。尽管C语言并没有强制要求用补码表示有符号整数，但它显然最适合使用这种表示形式。

当目标类型为_Bool时，所有非零的原值都被映射为1。只有原值0被转换为0。

浮点类型转换为整数类型 浮点类型到整数类型的转换应该产生一个尽可能与原对象相等的值。如果浮点数具有非零的小数部分，小数部分就会被丢弃。也就是说，这种转换一般涉及浮点数的截断。

如果浮点数甚至无法近似地用新类型来表示，转换的行为就是未定义的。例如，浮点数的主值太大或者把负的浮点数转换为一种无符号整数类型。上溢和下溢的处理常常由编译器实现者决定。

指针类型转换为整数类型 当原值是个指针并且目标类型不是_Bool时，这个指针就被当做一种无符号整数类型，它的长度与这个指针的长度相同。然后，这个无符号整数使用前面所列出的规则转换为目标类型。如果null指针不是用0表示的，当null指针转换为整数时必须显式地把它转换为0。

过去，C程序员假设指针可以转换为long类型并转换回来，而不会丢失信息。尽管这几乎总是正确的，但它并不是语言定义所要求的。在C99中，如果stdint.h头文件定义了intptr_t和uintptr_t类型，它们分别表示可以保存指针值的有符号和无符号整数类型。问题是，有些计算机所使用的指针类型的长度可能大于最长的整数类型。

参考：_Bool类型 第5.1.5节；字符类型 第5.1.3节；浮点类型 第5.2节；整数类型 第5.1节；intptr_t 第21.5节；溢出 第7.2.2节；指针类型 第5.3节；uintptr_t 第21.5节；stdint.h第21章；无符

号类型 第5.1.2节； void *类型 第5.3.1节。

6.2.4 转换为浮点类型

只有算术类型可以转换为浮点类型。

当float转换为double或double转换为long double时，转换结果的值与原值相同。这可以看成是浮点类型的表示形式选择上的限制。

当double转换为float或long double转换为double时，位于目标值范围内的原值用新类型表示，转换结果应该是最接近原值的两个浮点值之一。至于原值被截断还是四舍五入则取决于编译器。

如果原值超出了目标类型可以表示的值的范围，例如double值的主值太大或太小，无法用float表示时，转换结果就是未定义的，就像程序中所出现的上溢和下溢一样。

当整数类型转换为浮点类型时，如果整数值能够准确地用浮点类型表示，转换结果就是那个等值的浮点数。如果整数值无法准确地用浮点类型表示，但位于浮点类型可以表示的范围之内，则应该选择两个最接近的浮点值之一作为结果。如果整数值位于浮点类型可表示的值的范围之外，其结果是未定义的。

复数类型转换为浮点类型（C99） 当一种复数类型转换为另一种复数类型时，它的实部和虚部浮点成分根据（实数）浮点转换规则进行相应的转换。

当一种实数类型（整数或浮点数）转换为一种复数类型时，复数值的虚部被设置为0.0（如果可以的话用+0.0）。实数类型到复数类型的实部的转换遵循正常的从其他算术类型转换为浮点类型（实数）的规则。

当把复数类型转换为实数类型（浮点数或整数）时，丢弃虚数部分，而通过一般的规则把实部转换为（实数）浮点类型。_Imaginary类型（如果存在）表示实部总是为零的复数类型。从实数类型转换为虚数类型（或反之）所产生的结果总是0，这是它们唯一可以采用的共同值。从_Complex类型转换为_Imaginary类型时，将会丢弃实数部分。从_Imaginary类型转换为_Complex类型时，将把结果的实部设置为零。

参考：复数类型 第5.2.1节； 浮点类型 第5.2节； 整数类型 第5.1节； 溢出 第7.2.2节。

6.2.5 转换为结构和联合类型

不同的结构或联合类型之间不允许进行转换。

参考：结构类型 第5.6节； 联合类型 第5.7节。

6.2.6 转换为枚举类型

它与转换为整数类型的规则相同。一些允许的转换，如在枚举类型和浮点类型之间，可能是不好的编程风格的特征。

参考：枚举类型 第5.5节。

6.2.7 转换为指针类型

一般而言，指针和整数类型可以转换为指针类型。在一些特殊的情况下，数组或函数也可以转换为指针。

任何类型的null指针都可以转换为其他指针类型，并且仍然可以被当做null指针。在转换之

后，null指针的表示形式可能会发生变化。

对于任意的类型S和D，一个“指向S的指针”类型的值可以转换为“指向D的指针”类型。在标准C中，对象指针不能被转换为函数指针，反之亦然。但是，转换所产生的结果指针的行为可能受到表示形式改变以及编译器所施加的对齐限制的影响。

整数常量0或任何值为零的整数常量，或者转换为void *类型的这种常量，都可以表示null指针常量，并且总是可以转换为其他指针类型。这种转换的结果是与其他任何合法指针不同的null指针。不同指针类型的null指针可具有不同的内部表示形式。null指针并不一定要把它的所有位都设置为0。

除了常量0之外的整数也可以转换为指针类型，但转换结果是不可移植的。进行这种转换的意图是指针被认为是一种无符号整数类型（具有与指针类型相同的长度），并根据标准的整数转换规则把源类型转换为目标类型。

“T数组”类型的表达式被转换为“指向T的指针”类型的值，这个指针指向这个数组的第1个元素。这是寻常单目转换的一种（第6.3.3节）。

通过对函数替代指针，“返回T的函数”（例如，函数指定者）类型的表达式转换为“指向返回T的函数的指针”类型的值。这通常发生在单目转换部分中（第6.3.3节）。

参考：对齐限制 第6.1.3节；数组类型 第5.4节；函数调用 第7.4.3节；函数指定符 第7.1节；整数类型 第5.1节；指针类型 第5.3节；sizeof操作符 第7.5.2节；寻常单目转换 第6.3.3节。

6.2.8 转换为数组和函数类型

C不允许把任何类型转换为数组或函数类型。

例子

具体地说，C不允许数组类型之间以及函数类型之间的转换：

```
extern int f();
double d;
d = (( double () )f) ();      /* Invalid! */
d = (double) f();             /* OK */
d = (*(double (*)()) f)();
          /* Valid, but will have unexpected results */
```

在最后一条语句中，f的地址被转换为一个指向返回类型为double的函数的指针。然后，这个指针被解引用，并且这个函数被调用。这种做法是合法的，但最终存储在d中的值很可能是垃圾值，除非f确实（与以前的外部声明相比）被定义为返回一个double类型的值。

6.2.9 转换为void类型

任何值都可以转换为void类型。当然，这种转换的结果无法用来表示任何东西。这种转换可能发生在表达式的值将被丢弃的场合，例如在表达式语句中。

例子

把表达式转换void类型的最常用场合是忽略调用类型的结果。例如，调用printf函数的目的是向标准输出流写入一些信息。它返回一个错误提示，但这个提示常常被忽略。并没有必要把这个函数的调用结果转换为void类型，但这种做法明白地告诉读者程序员有意忽略这个函数的返回值。

```
(void) printf("Goodbye.\n");
```

参考：丢弃表达式 第7.13节；表达式语句 第8.2节；void类型 第5.9节。

6.3 寻常转换

6.3.1 类型转换

本章前面所讨论的所有转换都可以通过类型转换显式地进行，而不会出现错误。表6-2对允许进行的类型转换进行了总结。注意，标准C并不允许函数指针显式地直接转换为对象指针（或反之），尽管这种转换可以通过一种适当的整数类型来进行。这个限制反映了对象指针和函数指针在表示形式上可能具有很大的区别。

表6-2 允许进行的类型转换

目标（转换）类型	允许的源类型
任何算术类型	任何算术类型
任何整数类型	任何指针类型
指向（对象）T的指针，或(void *)	(a) 任何整数类型
	(b) (void *)
	(c) 对于任何Q，表示指向（对象）Q的指针
	(d) 对于任何Q，表示指向（函数）Q的指针[a]
指向（函数）T的指针	(a) 任何整数类型
	(b) 对于任何Q，表示指向（对象）Q的指针
	(c) 对于任何Q，表示指向（对象）Q的指针[a]
结构或联合	无，不允许这种类型转换
T类型的数组或返回T的函数	无，不允许这种类型转换
void	任何类型

a 标准C不允许这种转换。

类型限定符的存在与否并不会影响类型转换的有效性，有些类型转换可以绕过类型限定符。在赋值时允许进行的转换具有更强的限制性。

标准C保证一个指针对象转换为void *然后再转换回原来的类型后将保持它的原值。在其他C编译器中，通过char *进行这种转换具有相同的效果。

参考：赋值转换 第6.3.2节；类型转换 第7.5.1节；类型限定符 第4.4.3节；void * 第5.3.1节。

6.3.2 赋值转换

在简单的赋值表达式中，赋值操作符左边和右边的表达式的类型应该是相同的。如果不是，就会试图把赋值符右边的表达式的值转换为左边的类型。表6-3列出了合法的转换，它们是类型转换的一个子集。除非另有提示，ISO类型限定符的存在并不会影响转换的合法性，尽管const限定的左值无法出现在赋值符的左边。

表6-3 允许进行的类型转换

左边的类型	右边允许出现的类型
任何算术类型	任何算术类型
_Bool（C99）	任何指针类型
结构或联合类型[a]	兼容的结构或联合类型
(void *)[b]	(a) 常量0
	(b) 指向（对象）T_1的指针[c]
	(c) (void *)
指向（对象）T_1的指针[b,c]	(a) 常量0
	(b) 指向T_2的指针，其中T_1和T_2是兼容的
	(c) (void *)
指向（函数）F_1的指针[b]	(a) 常量0
	(b) 指针F_2的指针，其中F_1和F_2是兼容的

a 有些旧式的C编译器并不支持结构或联合的赋值

b 左边的引用类型必须具有右边的引用类型的所有限定符

c 如果另一个指针为void *类型，T_1可以是不兼容的类型（C99）。

如果不采用显式的转换，其他任何转换将被遵循ISO标准的C编译器所拒绝，但传统C编译器几乎总是允许混合指针类型的赋值，并常常允许在类型转换表达式中使用任何类型。

支配指针赋值的规则对于类型限制符提出了一些条件，因为赋值总是可以绕过限定符。把一个指针赋值给_Bool类型时，如果该指针为null，则所赋的值为0，否则为1。

参考：赋值操作符 第7.9.1节；类型转换 第6.3.1节；兼容类型 第5.11节。

6.3.3 寻常单目转换

寻常单目转换在执行一个操作之前确定是否并如何对它的唯一操作数执行转换。这类转换的目的是尽量减少操作符必须处理的算术类型的数量。对于单目操作符!、−、+、~和*，它们的操作数会自动进行转换。对于双目的<<和>>操作符，每个操作数会单独进行转换。

转换级别 由于C99增加了一些额外的整数类型，另外编译器可能会对类型集合进行扩展，因此很难简单而又精确地描述这些隐式类型转换。C99引入了转换级别的概念，帮助解释这些转换。我们将在这里使用转换级别。对于C89，可以简单地忽略long long、_Bool和其他扩展整数类型。对于传统C，可以参见本节后面的讨论。

转换级别是分配给每种整数类型的一个数值，指定了它的转换顺序。表6-4列出了标准整数类型可能的赋值顺序。这张表并没有列出枚举类型，但它们与底层整数类型的转换级别相同。

表6-4 转换级别

级 别	级别的类型
60	long long int、unsigned long long int（C99）
50	long int、unsigned long int
40	int、unsigned int
30	short、unsigned short
20	char、unsigned char、signed char
10	_Bool

表示级别的具体数字并没有什么讲究，但标准类型必须采用这张表所显示的相对数值顺序。没有选择连续整数的原因是C编译器可能在表中插入自己的扩展整数类型。扩展类型的转换级别必须遵循下面这些规则：它们的转换级别必须低于具有更高精度的类型，并且必须低于任何具有相同精度的标准类型。两个不同

的有符号整数类型不得拥有相同的转换级别，对于采用相同表示形式的无符号类型和有符号类型，它们的转换级别必须相同。

根据前面所列出的转换级别，表6-5列出了寻常单目转换。实际所执行的将是表中第1个起作用的转换。如果没有起作用的转换，就不会执行任何转换。作用于整数的单目转换称为整型提升。数组和函数类型的转换有时是被禁止的，参见第6.2.7节对例外情况的讨论。

表6-5 寻常单目转换（选择首先起作用的）

如果操作数类型如下	标准C把它转换为	传统C把它转换为
float	（不转换）	double
T数组	指向T的指针	（与标准C相同）
返回T的函数	指向返回T的函数的指针	（与标准C相同）
转换级别大于或等于int的整数类型[a]	（不转换）	（与标准C相同）
转换级别小于int的有符号类型	int	（与标准C相同）
转换级别小于int的无符号类型，并且它们的值可以用int类型表示	int	unsigned int
转换级别小于int的无符号类型，并且它们的值无法用int类型表示	unsigned int	（与标准C相同）

a int、signed int或unsigned int类型的位段的转换级别被认为小于int，这意味着它们的转换级别取决于它们的值是否都可以用int类型来表示。

例子

假设S是标准C的unsigned short类型的变量，并且它的值是1。如果short的范围小于int的范围，则表达式(－S)的类型是int，它的值为－1。但是，如果short的范围与int相同，这个表达式的类型就是unsigned，并且它的值是一个很大的正整数。这是因为在第1个例子中，S在应用单目负号操作符之前被提升为int类型。在第2种情况下，S被提升为unsigned类型。

如果是int、signed int或unsigned int类型的位段，它们的转换级别被认为低于int。

传统C编译器按照不同的方式执行这些转换。首先，所有具有较低转换级别的无符号类型被转换为unsigned int，因此就算没有保留操作数的值，至少保留了它的符号（程序员应该注意标准C的转换，因为提升结果的符号取决于编译器，可能会影响外周表达式的含义）。其次，float类型被转换为double类型，这样就减少了所需要的浮点型库函数的数量，虽然这种转换可能会造成一些性能的损失。在标准C中，这种权衡不再是强迫的，尽管编译器可以选择继续执行这些提升。

数组和函数的转换 寻常单目转换指定了数组类型的值被转换为指向数组第1个元素的指针，但下面这些情况则是例外：

1. 数组是sizeof或取地址操作符（&）的参数。
2. 用于初始化字符数组的字符串常量。
3. 用于初始化wchar_t类型的数组的宽字符串常量。

在C99中，这种转换在数组类型的任何值中都会出现。在C99之前，这种转换只是在数组类型的左值上执行的。

例子

```
char a[] = "abcd"; /* No conversion */
char *b = "abcd";  /* Array converted to pointer */
int i = sizeof(a); /* No conversion; size of whole array */
b = a + 1;         /* Array converted to pointer. */
```

寻常单目转换规定了函数指定符被转换为指向这个函数的指针，除非这个指定符是sizeof操作符或取地址操作符（&）的操作数（如果它是sizeof的操作数，这种用法就是非法的）。

例子

```
extern int f();
int (*fp)();
int i;
fp = f;          /* OK, f is converted to &f */
fp = &f;         /* OK, implicit conversion suppressed */
i = sizeof(fp);  /* OK, result is the size of the pointer */
i = sizeof(f);   /* Invalid */
```

参考：位反操作符~ 第7.5.5节；扩展整数类型 第5.1.4节；函数调用 第7.4.3节；函数指定符 第7.1节；间接访问操作符* 第7.5.7节；初始化值 第4.6节；逻辑反操作符！ 第7.5.4节；左值 第7.1节；移位操作符<<和>> 第7.6.3节；单目负号操作符- 第7.5.3节；宽字符串 第2.7.4节。

6.3.4 寻常双目转换

当两个值必须一起进行操作时，它们首先根据寻常双目转换规则转换为一种公共类型，它一般也就是结果的类型。在绝大多数双目操作符中，转换将作用于两个操作数，在条件操作符中，转换将作用于第2个和第3个操作数。寻常单目转换和寻常双目转换合称为寻常算术转换。

对两个操作数执行寻常双目转换的操作符首先独立地对每个操作数执行寻常单目转换，加宽较短的值，并把数组和函数转换为指针。在此之后，如果任一操作数不是算术类型或者两个操作数属于相同的算术类型，则不再执行进一步的转换。否则，就会对两个操作数执行表6-6中第1个可以应用的转换。这张表假定两个操作数都不是复数类型。读者可以参考后面对处理复数操作数的讨论。

表6-6 寻常双目转换（选择首先起作用的）

如果任一操作数的类型是[a]	另一操作数的类型是[a]	标准C把它们转换为	传统C把它们转换为
long double	任何实数类型	long double	无应用
double	任何实数类型	double	（与标准C相同）
float	任何实数类型	float	double
任何无符号类型	任何无符号类型	具有更高转换级别的无符号类型	（与标准C相同）
任何有符号类型	任何有符号类型	具有更高转换级别的有符号类型	（与标准C相同）
任何无符号类型	转换级别更小或相同的有符号类型	该无符号类型	（与标准C相同）
任何无符号类型	具有更高转换级别的有符号类型，并且可以表示该无符号类型的所有值	该有符号类型	该有符号类型的无符号版本
任何无符号类型	具有更高转换级别的有符号类型，并且无法表示该无符号类型的所有值	该有符号类型的无符号版本	（与标准C相同）
任何其他类型[b]	任何其他类型	（无转换）	（与标准C相同）

a 这个规则假设寻常单目转换已经作用于每个操作数。

b 复数操作数在正文中讨论。

例子

当long操作数和unsigned操作数一起出现时（并且long类型严格地大于unsigned类型），标准C的规则与传统C的规则不同。下面这个程序判断将发生哪种转换：

```
unsigned int UI = -1;
long int LI = 0;
int main()
{
    if (UI < LI) printf("long+unsigned==long\n");
    else printf("long+unsigned==unsigned\n");
    return 0;
}
```

复数类型和寻常双目转换 在C99中，在寻常双目转换中必须考虑复数类型。在实数和复数的混合运算中，由于性能原因，实数类型的操作数并不会转换为复数类型。但是，编译器仍然会执行一些转换，使两个操作数具有相同的浮点精度。然后，在处理实数和复数的混合运算时，一般把实数操作数看成已经转换成为复数类型（当然，编译器也可以实际执行这种假设的转换）。这种运算的结果类型是通过转换之后的复数操作数的类型。

具体地说，如果两个操作数都是复数类型，则更短的那个操作数被转换为较长操作数的类型，它也是结果的类型。这个行为对应于两个浮点实数操作数的转换方式。

当一个操作数是复数类型，另一个操作数是整数时，整数操作数就被转换为与复数类型对应的浮点实数类型。例如，如果复数操作数的类型是float _Complex，则这个整数应该转换为float类型。这种运算的结果是复数操作数的类型。

当一个操作数是复数类型，另一个是浮点实数类型时，较小精度的那个类型就会被转换（在实数或复数定义域内），使它转换到与另一类型的精度相同。例如，当float类型与double _Complex类型一起出现时，float操作数就被提升为double类型。当long double与double _Complex一起出现时，double _Complex就会提升为long double _Complex。

6.3.5 默认的函数实参转换

如果表达式作为函数调用的实参，并且这个函数并没有原型，或者当这个表达式作为原型实参列表的“...”部分的实参，则表达式的值在传递给函数之前将进行转换。默认的函数实参转换与寻常单目转换相同，区别在于float类型的实参总是会被转换为double类型，即使在标准C中也是一样。

如果被调用的函数具有原型，则它的实参不需要执行寻常整型提升，float类型的实参也不必转换为double类型。如果编译器需要，也可以执行这些转换。但是，这些规则允许编译器对调用序列进行优化。另外，数组和函数仍然会转换为指针。

在C99的原型中，如果一个数组类型的形式参数具有类型限定符列表L（位于方括号[和]中），则实际的数组实参就被转换为L限定的指向元素类型的指针。这个问题将在第9.3节进行更深入的讨论。

float到double的实参转换可以帮助以前版本的传统C和标准C控制库函数的数量，因为它可以避免同时拥有float和double版本的库函数。C99指定了一组丰富的数学函数，除了double类型的之外，还包括float和long double类型的。

参考：数组限定符列表 第4.5.3节；函数调用 第7.4.3节；数学函数 第17章；原型 第9.2节；寻常单目转换 第6.3.3节。

6.3.6 其他函数转换

函数的形式参数的声明类型以及返回值的类型受一些类型调整的影响，其方式与函数实参的转换相同。第9.4节将讨论这个问题。

6.4 C++兼容性

赋值转换

在C++中，必须通过强制类型转换把void *类型的指针转换为其他类型的指针。在C中，也可以使用强制类型转换。但是在赋值时，并不需要采用强制类型转换。

例子

malloc函数返回一个void *类型的指针，指向一块新分配的内存区域。

```
#include <stdlib.h>
char * cp;
const int SIZE = 10 * sizeof(char);
...
cp = malloc(SIZE);              /* OK in C, not C++ */
cp = (char *) malloc(SIZE);     /* OK in both */
```

另外，如果不使用强制类型转换，只有指向未限定对象（不是const或volatile）的指针可以转换为void *类型的指针。

例子

```
char * cp;
const char * const_cp;
void * vp;
...
vp = cp;                  /* valid in both C and C++ */
vp = const_cp;            /* valid in C, not in C++  */
vp = (void *) const_cp;   /* valid in both C and C++ */
```

参考：赋值转换 第6.3.2节。

6.5 练 习

1. 下面这张表列出了在类型转换中所使用的源类型和目标类型组合的列表。哪些转换是标准C所允许的？哪些是传统C所允许的（对于传统C，把void替换为char）？

目标类型	源类型
(a) char	int
(b) char *	int *
(c) int (*f) ()	int *
(d) double *	int
(e) void *	int (*f)()
(f) int *	t * （其中：typedef int t）

2. 在练习1的表格中，哪几对是标准C的赋值转换所允许的？哪几对是传统C的赋值转换所允许的（目标类型是左边的类型，源类型是右边的类型）？
3. 当传统C的寻常双目转换作用于下面这几对类型时，结果类型是什么？在哪几种情况下，结果与标准C不同？
 (a) char 和 unsigned　　(d) char 和 long double
 (b) unsigned 和 long　　(e) int [] 和 int *
 (c) float 和 double　　(f) short () 和 short ()
4. 是否允许C编译器让char类型表示从−2 147 483 648到2 147 483 647的值？如果可以，在这种编译器中，sizeof(char)的值是什么？int类型的最小范围和最大范围又分别是什么？
5. sizeof(long double)和sizeof(int)?之间必须具有什么关系？
6. 假设计算机A和B都采用字节编址，并且都采用32位（4个字节）的字长。但是，计算机A采用大端字节顺序，计算机B采用小端字节顺序，整数128在计算机A中存储在一个字中，并传输到计算机B的一个字中。方法是把计算机A的字的第1个字节复制到计算机B的字的第1个字节，接下来以此类推。当传输完成时，存储在计算机B的字中的整数值是什么？如果A采用小端字节顺序而B采用大端字节顺序，其结果又是什么？

第7章 表达式



C语言提供了丰富的操作符，可以访问底层硬件所提供的大多数操作。本章讨论表达式的语法，并描述每个操作符的功能。

7.1 对象、左值和指示符

对象（object）是一块内存区域，可以读取它的值或者向它存储数据。*左值*（lvalue）是一种表达式，可以读取或修改它所引用的对象。只有左值表达式可以作为赋值操作符的左操作数。不属于左值的表达式有时称为*右值*（rvalue），因为它只能出现在赋值操作符的右边。左值可以是对象或不完整类型，但不能是void类型。

当标准C使用左值这个术语时，并不保证允许对它所指定的对象进行修改。如果左值是数组类型、不完整类型或const限定的类型，或者是部分成员具有const限定符的结构或联合类型（递归地应用于嵌套的结构和联合），这种左值就不允许被修改。可修改的左值（modifiable lvalue）这个术语强调允许对左值所指示的对象进行修改。

函数指示符（function designator）是个函数类型的值。它既不是对象也不是左值。函数的名称是函数指示符，对函数指针进行解引用的结果也是函数指示符。C常常以不同的方式看待函数和对象，所以我们需要仔细区分“函数类型”和“对象类型”、“左值”和“函数指示符”以及“函数指针”和“对象指针”。“指定了一个对象的左值”这句话是冗余的，但我们可以在适当的时候用它来强调不包括函数指示符。

表7-1列出了可以作为左值的C表达式，以及使这些表达式成为左值所必须满足的特殊条件。其他形式的表达式都不能作为左值。而且，除了字符串常量之外，这里所列出的表达式如果是数组类型，也不能作为左值。不能作为左值的表达式包括：数组名、函数、枚举常量、赋值表达式、类型转换表达式和函数调用。

表7-1 可以作为左值的非数组表达式

表达式	额外的要求
name	name必须是个变量
e[k]	无
(e)	e必须是个左值
e.name	e必须是个左值
e->name	无
*e	无
字符串常量	无

表7-2列出了操作数必须是左值的操作符。

表7-2 需要左值操作数的操作符

操作符	要求
&（单目）	操作数必须是左值或函数名
++ --	操作符必须是左值（后缀和前缀形式）
= += -= *= /= %= <<= >>= &= ^= \|=	左操作数必须是左值

参考：取地址操作符 第7.5.6节；赋值表达式 第7.9节；类型转换表达式 第7.5.1节；成员选

择 第7.4.2节；减值表达式 第7.4.4节、第7.5.8节；枚举 第5.5节；函数调用 第7.4.3节；增值表达式 第7.4.4节、第7.5.8节；间接访问表达式 第7.5.7节；字面值 第2.7节；名称 第7.3.1节；字符串常量 第2.7.4节；下标 第7.4.1节。

7.2 表达式和优先级

本章所描述的表达式的语法完整地指定了C的操作符的优先级。为了对这些信息进行总结，表7-3包含了C操作符的简洁列表，按照优先级从高到低的顺序排列，同时列出了这些操作符的结合性。

表7-3 C操作符的优先级

标记	操作符	类型	优先级	结合性
名称、字面值	简单标记	基本	16	无
a[k]	下标	后缀	16	从左到右
f(...)	函数调用	后缀	16	从左到右
.	直接选择	后缀	16	从左到右
->	间接选择	后缀	16	从左到右
++ --	增值、减值	后缀	16	从左到右
(类型名){init}	复合字面值（C99）	后缀	16	从左到右
++ --	增值、减值	前缀	15	从右到左
sizeof	长度	单目	15	从右到左
~	位非	单目	15	从右到左
!	逻辑非	单目	15	从右到左
- +	算术负、正号	单目	15	从右到左
&	取地址	单目	15	从右到左
*	间接访问	单目	15	从右到左
(类型名)	转换类型	单目	14	从右到左
* / %	乘除模	双目	13	从左到右
+ -	加减	双目	12	从左到右
<< >>	左右移位	双目	11	从左到右
< > <= >=	关系	双目	10	从左到右
== !=	相等/不相等	双目	9	从左到右
&	位与	双目	8	从左到右
^	位异或	双目	7	从左到右
\|	位或	双目	6	从左到右
&&	逻辑与	双目	5	从左到右
\|\|	逻辑或	双目	4	从左到右
?:	条件	三目	3	从右到左
= += -= *= /= %= <<= >>= &= ^= \|=	赋值	双目	2	从右到左
,	逗号	双目	1	从左到右

7.2.1 操作符的优先级和结合性

C的每个表达式操作符具有优先级和结合性规则。如果表达式并没有使用括号对操作符的操作数进行明确的分组，操作数就根据操作符的优先级进行分组。如果两个操作符具有相同的优

先级，则操作数根据操作符的左结合性或右结合性组合到左边或右边的操作符。具有相同优先级的操作符总是具有相同的结合性。

优先级和结合性规则决定了表达式的含义，但并没有指定大型表达式或语句的子表达式在运行时的求值顺序。第7.12节将讨论求值的顺序。

例子

下面是一些优先级和结合性规则的例子：

原表达式	等价表达式	等价的原因
a * b + c	(a * b) + c	*的优先级高于+
a += b \| = c	a += (b \| = c)	+=和\|=具有右结合性
a − b + c	(a − b) + c	-和+具有左结合性
sizeof(int) *p	(sizeof(int)) *p	sizeof的优先级高于类型转换
*p -> q	* (p -> q)	->的优先级高于*

对结合性规则总结如下：除了赋值操作符具有右结合性之外，其余的双目操作符都是左结合性的。条件操作符也是右结合性。单目和后缀操作符有时被描述为具有右结合性，但这只是为了描述像*x++这样的表达式将解释为*(x++)而不是(*x)++。我们倾向于简单地声称后缀操作符比（前缀）单目操作符具有更高的优先级。

参考：赋值操作符 第7.9节；双目操作符 第7.6节；字符串连接 第2.7.4节；条件操作符 第7.8节；后缀操作符 第7.4.4节；单目+ 第7.5.3节。

7.2.2 溢出和其他算术异常

对于C的某些操作，例如加法和乘法，它们的算术结果可能无法用预期的结果类型（由寻常转换规则所决定）的值表示。这种情况称为溢出（有时又区分上溢和下溢）。

一般而言，C语言并没有指定溢出的后果。一种可能性是产生（正确类型的）不正确的值；另一种可能性是程序将会终止执行；第三种可能性是程序检测（其方式因编译器而异）到某种因计算机而异的陷阱或异常，并对它们进行处理。

对于某些操作，C语言明确规定了某些操作数所产生的结果是不可预测的，或者更严格地说，它总是会产生一个值，但这个值对于某些操作数而言是不可预测的。如果除法操作符/或求余操作符%的右操作数是0，则运算结果是不可预测的。如果移位操作符<<或>>的右操作数太大或者是负数，就会产生一个无法预料的值。

按照传统，C的所有实现都忽略了有符号整数的溢出问题，因为运算结果总是由实现操作的机器指令所产生的（许多使用补码表示有符号整数的计算机简单地通过产生真实补码结果的低端位来处理加法和减法的溢出。毫无疑问，很多现有的C程序依赖这种行为，但这种代码从技术上来说是不可移植的）。浮点数的上溢和下溢通常是由机器所支持的一种方便方法来处理。如果机器体系结构提供了超过一种的方式处理异常的浮点结果，C程序员就可以通过库函数使用这些方法。

对于无符号整数，C语言对溢出问题有一些特殊的规定：无符号整数的每次操作总是会产生一个结果值，是同余模数2^n减去该操作的真正数学结果之差（其中n是表示这个无符号结果的位数）。它相当于计算真正结果的正好低端n位的结果（如果真正结果是负数，例如一个较小的无

符号整数减去一个较大的无符号整数，其结果就是真正的补码结果）。

例子

作为一个例子，假设无符号类型的对象是用16位表示的，则无符号整数值4减去无符号整数值7将产生无符号值65 533（$2^{16}-3$），因为这个值是同余模数2^{16}减去真正的运算结果−3的结果。

这个规则的一个重要后果是，如果两种计算机所使用的表示形式具有相同的位数，则无符号整数的运算在这两种计算机之间就是完全可以移植的。我们可以很方便地用较少数量的位来模拟其他计算机上的无符号运算。

参考：除法操作符/ 第7.6.1节；浮点类型 第5.2节；余数操作符% 第7.6.1节；移位操作符<<和>> 第7.6.3节；有符号类型 第5.1.1节；无符号类型 第5.1.2节。

7.3 基本表达式

基本表达式共有3种：名称（标识符）、字面值常量和带括号的表达式：

基本表达式：

标识符
常量
带括号的表达式

在C中，函数调用、下标表达式和成员选择表达式按照传统都被列为基本表达式。但是，我们将在下一节介绍后缀表达式时讨论它们。

7.3.1 名称

名称的值取决于它的类型。名称的类型是由它的声明（如果有）决定的，如第4章所述。

当一个变量名被声明为算术、指针、枚举、结构或联合类型时，它的求值结果就是一个这种类型的对象；名称是个左值表达式。枚举常量名称的求值结果是相关联的整数值，它并不是左值。

例子

在下面这个例子中，4种颜色名称是枚举常量。switch语句（在第8.7节描述）根据参数color的值在4条语句中选择其中之一执行：

```
typedef enum { red, blue, green } colortype;

colortype next_color(colortype color)
{
    switch (color) {
        case red    : return blue;
        case blue   : return green;
        case green  : return red;
    }
}
```

数组名的求值结果就是该数组，它是个左值，但不能被修改。除非数组是sizeof操作符、取地址操作符（&）的参数，或者是个用字符串常量进行初始化的字符数组，否则它将被转换成指向数组第1个元素的指针，这是寻常单目转换的结果。

例子

当数组名是sizeof操作符的参数时，并不会发生从数组名到指针的转换。因此，sizeof表达式的结果是数组的长度而不是指针的长度。

```
extern void PrintMatrix();
int Matrix[10][10], total_length, row_length;

total_length = sizeof Matrix;
row_length = sizeof Matrix[0];
PrintMatrix(Matrix); /* pointer to first
                        element is passed */
```

函数名的求值结果就是该函数，它并不是左值。除非函数名作为取地址操作符（&）或sizeof操作符的参数，否则它将被转换为指向函数的指针，这是寻常单目转换的结果。&f的结果是指向f的指针，而不是指向f的指针的指针，sizeof(f)是非法的。

例子

这个例子显示了一个函数名作为另一个函数的参数：

```
extern void PlotFunction(double (*f)(double),
                         double x0, double x1);

double fn(double x) { return x * x - x; }

int main(void)
{   ...
    PlotFunction(fn, 0.01, 100.0); /* fn converts to &fn */
    ...
}
```

当名称作为表达式时，它无法引用语句标签、typedef名称、结构成员名称、联合成员名称、结构标签、联合标签或枚举标签。用于这些用途的名称位于不同的名字空间，与出现在表达式中的名称不同。这类名称有些可以通过特殊的结构在表达式中使用。例如，结构和联合成员的名称可以使用.或->操作符，typedef名称可以在类型转换表达式中使用，或者作为sizeof操作符的参数。

参考：数组类型 第5.4节；类型转换 第7.5.1节；枚举类型 第5.5节；函数调用 第7.4.3节；函数类型 第5.8节；左值 第7.1节；名字空间 第4.2节；选择操作符.和-> 第7.4.2节；sizeof操作符 第7.5.2节；typedef名称 第5.10节；寻常单目转换 第6.3.3节。

7.3.2 字面值

字面值（词法常量）是一种数值常量，当它作为表达式求值时，其结果是一个常量。除了字符串常量之外，字面值表达式不能作为左值。关于字面值以及它们的类型和值的讨论，请参见第2.7节。

7.3.3 带括号的表达式

带括号的表达式由左括号、表达式和右括号组成：

带括号的表达式：

(表达式)

带括号的表达式的类型与括号所包围的表达式的类型相同，不会执行转换。带括号的表达

式的值就是括号所包围的表达式的值，如果这个表达式是个左值，那么这个带括号的表达式也是个左值。括号并不保证实现特定的求值顺序（参见第7.12节）。

带括号的表达式可能只是出于分组的目的，简单地对被包围的表达式进行定界，或者是为了改变操作符的求值顺序，或者是为了使代码看上去更清楚。

例子

```
x1 = (-b + discriminant)/(2.0 * a)
```

参考：左值 第7.1节。

7.4 后缀表达式

后缀表达式共有6种：下标表达式、两种形式的成员选择（直接和间接）、函数调用以及后缀形式的增值和减值表达式。

后缀表达式：

基本表达式
下标表达式
成员选择表达式
函数调用
后缀增值表达式
后缀减值表达式
复合字面值 (C99)

函数调用、下标表达式和成员选择表达式在传统上列为基本表达式，但它们的语法更接近于后缀表达式。

7.4.1 下标表达式

下标表达式由后缀表达式、左方括号、任意表达式和右方括号组成。这种结构用于数组下标，其中后缀表达式（通常称为数组名）的求值结果是指向数组起始位置的指针，另一个表达式则是个整型偏移值：

下标表达式：

后缀表达式[表达式]

在C语言中，根据定义，表达式$e_1[e_2]$准确地对应于表达式*((e_1) + (e_2))。下标表达式的两个操作数将会执行寻常双目转换，其结果总是个左值。间接访问操作符（*）的操作数必须是个指针。要想让+操作符的结果是个指针，唯一的方法是让它的一个操作数是个指针，另一个操作数是个整数。因此，它要求表达式$e_1[e_2]$的其中一个操作数是指针，另一个操作数是整数。按照约定，e_1是数组名，e_2是整数表达式，但e_1也可以是指针，并且这两个操作数的顺序可以颠倒。下标的这个定义所导致的结果是数组的索引从0开始。

为了引用多维数组，需要使用复合的下标操作符。

例子

```
char buffer[100], *bptr = buffer;
int i = 99;
…
buffer[0] = '\0';       /* subscripting an array */
```

```
bptr[i-1] = bptr[0];    /* subscripting a pointer */
i[bptr]   = '\0';       /* unconventional subscripting */
```

包含100个元素的数组buffer的第1个元素用buffer[0]表示，最后一个元素用buffer[99]表示。buffer和bptr都是指向同一位置的指针，即buffer[0]，buffer数组的第1个元素。它们可以按照相同的方式在下标表达式中使用。但是，bptr是个变量（左值），因此可以让它指向其他地方：

```
bptr = &buffer[6];
```

在这个赋值之后，bptr[-4]所指向的位置和buffer[2]相同（这个例子说明了在有些情况下，负的下标也是合理的）。另外，也可以使用赋值表达式使bptr根本不指向任何位置：

```
bptr = NULL;   /* Store a null pointer into bptr. */
```

但是，数组名buffer不是个左值，无法被修改。当它作为指针时，它总是指向一个固定的位置，就像它是按照下面的方式声明的：

```
char * const buffer;
```

例子

下面的代码在一个10 × 10的数组matrix的对角线元素中存储1.0，在其他元素中存储0.0：

```
int matrix[10][10];
...
for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
        matrix[i][j] = ((i == j) ? 1.0 : 0.0);
```

在下标括号中使用逗号表达式是不良的编程风格，因为它容易误导熟悉其他编程语言的读者，以为它表示的是多维数组的下标。

例子

下面这个表达式

```
commands[k=n+1, 2*k]
```

容易让人以为它表示二维数组commands的一个元素，其中两个下标表达式分别是k=n+1和2*k。但是在C中，它实际上是一维数组commands的一个元素，下标表达式的结果是2*k（在k已经被赋值为n+1之后）。如果确实需要使用逗号表达式（很难想出这样的例子），应该把整个逗号表达式放在括号内，以表示这种不寻常的用法。

```
commands[(k=n+1, 2*k)]
```

使用指针和类型转换，把一个多维数组当成一维数组是可能的。有时为了提高效率，可以采用这种做法。必须记住，C的数组是按照行主序存储的。

例子

下面的代码建立了一个单位矩阵，即对角线元素为1且其他元素为0的矩阵。这段代码看上去有点晦涩，但它的速度很快。它把一个二维数组看成是个具有相同数量的元素的一维向量，因此简化了下标，并消除了嵌套循环的需要。

```
#define SIZE 10
double matrix[SIZE][SIZE];
int i;
for (i = 0; i < SIZE*SIZE; i++)
    ((double *)matrix)[i] = 0.0;    /* zero all elements */
```

```
for (i = 0; i < SIZE*SIZE; i += (SIZE + 1))
   ((double *) matrix)[i] = 1.0; /* set diagonals to 1 */
```

参考：加法操作符+ 第7.6.2节；数组类型 第5.4节；逗号表达式 第7.10节；间接访问操作符* 第7.5.7节；整数类型 第5.1节；左值 第7.1节；指针类型 第5.3节。

7.4.2 成员选择

成员选择操作符用于访问结构和联合类型的字段（成员）：

成员选择表达式：

直接成员选择
间接成员选择

直接成员选择

后缀表达式.标识符

间接成员选择

后缀表达式->标识符

直接访问表达式由后缀表达式、点号（.）和标识符组成。后缀表达式必须是结构或联合类型，标识符必须是该类型的一个成员的名称。选择表达式的结果是结构或联合中以这个标识符命名的成员。

如果结构或联合类型是个左值，直接成员选择表达式的结果是个左值（只有由函数所返回的结构或联合值不是左值）。如果结果是个左值并且被选择的成员并不是数组，它就可以被修改。

例子

```
struct S {int a,b;} x;
extern struct S f(); /* structure-returning function */
int i;
…
x = f();       /* OK */
i = f().a;     /* OK */
f().a = i;     /* Invalid; f() is not an lvalue */
```

（最后一个赋值即使是合法的，也是不合理的。函数f将返回某个结构的一份拷贝，可能有某个成员的值进行了修改，就在语句结束时整份拷贝被丢弃之前。）

（有些非标准的C编译器并不允许函数返回结构。在那些允许函数返回结构的编译器中，有些编译器不允许对它使用选择操作符，它们把f().a看成是错误的。）

如果点号之前的表达式具有类型限定符，或者它的成员具有类型限定符，则最终结果具有这些限定符的并集。

例子

下面这个赋值是非法的，因为x.a的类型是const int。这个const是从x继承而来的：

```
const struct {int a,b;} x;
…
x.a = 5;  /* Invalid */
```

间接成员选择表达式由后缀表达式、操作符->和名称组成。后缀表达式的值必须是指向结构或联合类型的指针，名称必须是这个结构或联合类型的成员名称。间接成员选择表达式的结果

是结构或联合类型中由名称所指定的成员，并且是个左值。除非这个左值是个数组，否则它是可以修改的。按照定义，表达式e -> name精确地对应于表达式(*e).name。

例子

在下面的代码中，Point结构的两个成员均按照不同的方式初始化为0.0，说明了上面这两种表达式是等价的：

```
struct {float x, y; } Point, *Point_Ptr;
…
Point.x = 0.0;                  /* Sets x to 0.0 */
Point_Ptr = &Point;
Point_Ptr->y = 0.0;    /* Sets y to 0.0 */
```

如果->之前的表达式具有类型限定符，或者它的成员具有类型限定符，则间接成员选择表达式所产生的结果具有这些限定符的并集。

有些C编译器允许间接访问操作符的左边使用null指针。对结果使用取地址操作符&，并把结果转换为整数类型，产生一个成员在这个结构中的偏移量（以字节计）。标准C并没有明确允许或禁止这种做法，但这种方式常常可行。

例子

```
#define OFFSET(type,field)  \
   ((size_t)&((type *)0)->field)
```

这个OFFSET宏与stddef.h头文件中的offsetof宏相似。

参考：取地址操作符& 第7.5.6节；间接访问操作符* 第7.5.7节；左值 第7.1节；offsetof宏 第11.1节；size_t 第13.1节；结构类型 第5.6节；类型限定符 第4.4.3节；联合类型 第5.7节。

7.4.3 函数调用

函数调用由后缀表达式（函数表达式）、左括号、由逗号分隔的表达式序列（实参表达式，可以为空）以及右括号组成：

函数调用：

后缀表达式(表达式列表$_{opt}$)

表达式列表：

赋值表达式
表达式列表，赋值表达式

函数表达式经过寻常单目转换之后，它的类型必须是“指向返回T的函数的指针”，其中T是某种类型。函数调用的结果为类型T，它不能作为左值。如果T是void，则这个函数调用不返回结果，可能无法用于在调用之后需要产生一个结果的上下文环境中。T不能是数组类型。

在标准C之前的编译器中，函数调用要求是“返回T的函数”类型，因此必须显式地对函数指针进行解引用。也就是说，如果fp是个函数指针，它所指向的函数只能通过(*fp)(...)的形式进行调用。如果fp是个形式参数，则可以作为例外，此时可以采用fp(...)的形式。

在执行函数调用时，首先对函数和实参表达式进行求值，C并没有指定它们的求值顺序。

接着，如果函数调用是根据标准C的原型（第9.2节）进行的，则实参表达式的值就被转换为原型所指定的对应形式参数的类型。如果无法进行这种转换，这个函数调用就是非法的。如

果函数具有可变数量的参数，则额外的参数根据寻常参数转换规则（第6.3.5节）进行转换，并且不会再对这些额外参数执行进一步的检查。

如果函数调用不是根据原型进行的，实参表达式只是根据寻常参数转换规则进行转换，编译器无需执行进一步的检查。这是因为在没有原型的情况下，编译器无法知道外部函数的形式参数的信息。

在实际参数被求值和转换之后，它们就被复制到被调用函数的形式参数。因此，所有的参数都是按照传值方式传递的。在被调用函数中，形式参数的名称是左值，但对形式参数的赋值只会修改形式参数中的那份拷贝，对实际参数（可能也是左值）并不会产生任何影响。

例子

考虑下面这个函数square，它返回实参的平方值：

```
double square(double y) { y = y*y; return y; }
```

假设x是个double类型的变量，它的值为4.0，并且执行函数调用square(x)。这个函数将返回16.0这个值，但x的值仍然为4.0。在square函数中对y的赋值只影响实际参数的那份拷贝。

只有当调用者的数据在被调用的函数中独立可见（例如是全局变量）或者调用者把指向数据的指针作为实参传递给被调用函数时，被调用函数才能修改调用者的数据。当指针作为实参传递给被调用函数时，指针值被复制，但它所指向的对象并没有被复制。因此，当被调用函数通过被复制的指针按照间接访问的方式对它所指向的对象进行修改时，调用者也能看到这种修改。

例子

下面这个swap函数交换两个整数对象的值，指向这两个对象的指针作为实参传递给这个函数：

```
void swap(int *xp, int *yp)
{
    int t = *xp;
    *xp = *yp;
    *yp = t;
}
```

如果a是所有元素均为0的整型数组，i是个值为4的整型变量，则在调用swap(&a[i],&i)之后，i的值是0，a[4]的值将是4。

数组类型的形式参数和实际参数总是被C转换为指针。因此，在函数中对数组形参进行修改将会影响实际参数，尽管看上去似乎并不是很直观。

例子

考虑下面这个函数f，它具有一个数组形参：

```
void f(int a[10])
{
    a[4] = 12;  /* changes caller's array */
}
```

如果vec是个整型数组，则调用f(vec)将把vec[4]设置为12。数组形参所指定的长度10并没有意义。a可以被声明为int a[]。

如果一个返回类型不是void的函数在函数的返回值将被丢弃的上下文环境中被调用，编译器将会对此发出警告。但是，丢弃像printf这样的非void函数的返回值是极为常见的做法，所以许

多程序员认为这种警告很讨厌。

例子

为了明确表示需要丢弃函数调用的结果，可以使用类型转换，例如像下面调用strcat一样：

```
(void) strcat(word, suffix);
```

如果逗号表达式被封装在一对括号中，它也可以作为函数的参数，这样就不会被误认为是多个不同的参数。

例子

假设希望在C程序中追踪对函数f的所有调用。如果f接受单个参数，则下面这个宏将在每次调用f之前插入对tracef的调用。

```
#define f(x) (tracef(__FILE__, __LINE__), f((x)))
```

如果对f的调用作为函数的实参，例如g(f(y))，则传递给g的实参就是个逗号表达式。

参考：实参和形参的一致 第9.6节；逗号操作符 第7.10节；丢弃表达式 第7.13节；函数类型 第5.8节；函数原型 第9.2节；间接访问操作符* 第7.5.7节；左值 第7.1节；宏展开 第3.3.3节；指针类型 第5.3节；printf 第15.11节；strcat 第13.1节；寻常实参转换 第6.3.5节；void类型 第5.9节。

7.4.4 后缀增值和减值操作符

后缀操作符++和--分别把它们的操作数加1和减1，并返回操作数的原值作为结果。它们是会产生副作用的操作符：

后缀增值表达式：

后缀表达式 ++

后缀减值表达式：

后缀表达式 --

这两个操作符的操作数都必须是可以修改的左值，可以是任何算术类型或指针类型。如果是++，操作数的值就加上常量1。如果是--，操作数的值就减去常量1。经过操作之后，操作数的值被修改，并且返回它们被修改之前的值。返回结果不再是左值。在执行加法或减法之前，会对操作数和常量1执行寻常双目转换。把被修改的值存储回操作数时，会执行寻常赋值转换。结果的类型是在转换之前左值操作数的类型。

例子

如果i和j都是整型变量，则语句i = j--;可以被改写为下面这两条语句：

```
i = j;
j = j-1;
```

如果发生了溢出并且操作数是有符号整数或浮点数，这些操作可能会产生不可预料的结果。对无符号类型的最大可表示值进行增值的结果是0，对无符号整数0进行减值的结果是这种无符号类型可以表示的最大值。

如果操作数是个指针，例如是“指向T的指针”（其中T是某种类型），++的效果相当于把这个指针从它当前所指向的对象前进一步，就像把指针移动到T类型的数组中的下一个元素（在字节编址的计算机上，这意味着把指针向前移动sizeof(T)个字节）。类似，--的效果是把指针后退一步，就像把指针移动到T类型数组中的前一个元素。不论是哪种情况，这种表达式的值是移动

之前的那个指针。

例子

在扫描数组或字符串的元素时，使用后缀增值操作符是极为常见的。例如，下面这个例子对一个字符串中的字符进行计数：

```
int string_length(const char *cp)
{
    int count = 0;
    while (*cp++) count++;
    return count;
}
```

参考：加法 第7.6.2节；数组类型 第5.4节；赋值转换 第6.3.2节；浮点类型 第5.2节；整数类型 第5.1节；左值 第7.1节；溢出 第7.2.2节；指针类型 第5.3节；标量类型 第5章；有符号类型 第5.1.1节；减法 第7.6.2节；无符号类型 第5.1.2节；寻常双目转换 第6.3.4节。

7.4.5 复合字面值

C99引入了复合字面值的概念，用于表示聚合类型的无名常量。复合字面值由带括号的类型名以及位于一对花括号内的初始化值列表组成。在初始化值列表的最后可以有个可选的逗号。

复合字面值：

（类型名称）{初始化值列表 $,_{opt}$ }　　（C99）

复合字面值创建了一个指定类型的无名对象，并返回一个该类型的左值。类型名称可以是任何对象类型或未知长度的数组类型。复合字面值中不能使用可变长度的数组，因为它们无法被初始化。结构、联合、数组和枚举类型最适合作为复合字面值。初始化值列表的格式和含义与相同类型和范围的对象声明中所允许的初始化值列表相同。这意味着复合字面值的未初始化成员将被默认初始化为0（参见第4.6节）。

可以在复合字面值的类型名称中使用const限定符，创建只读的字面值。如果未使用这个限定符，字面值就是可以修改的。如果两个只读的字面值具有相同的类型和值，则编译器可以选择让它们复用同一个存储空间。也就是说，它们的地址可能相同，就像相同的字符串常量一样。

例子

使Temp1指向一个可修改的字符串，使Temp2指向一个只读的字符串：

```
char *Temp1 = (char []){"/temp/XXXXXXXX"};
char *Temp2 = "/temp/XXXXXXXX";
```

POW2函数通过表格查找的方式计算一个较小整数的2次方：

```
inline int POW2(int n)
{
    assert( n >= 0 && n <= 7 );
    return (const int []){1, 2, 4, 8, 16, 32, 64, 128}[n];
}
```

DrawTo函数接受一个按值传递的Point结构，DrawLine则接受两个Point对象的地址为参数。

```
DrawTo( (struct Point){.x=12, .y=n+3} );
DrawLine( &(struct Point){x,y}, &(struct Point){-x,-y} );
```

如果一个复合字面值出现在一个文件的顶层，则这个无名对象具有静态范围，它在程序的整个执行期间一直存在。在这种情况下，初始化值列表只能包含常量值。如果复合字面值出现在函数中，则它具有自动范围，它的作用域就是包含它的最内层代码块。在接受复合字面值的地址为参数时，它的生命期是极为重要的，程序员必须保证在离开了这个字面值的作用域之后不再使用它。

每次进入包含复合字面值的代码块时，都会为这个字面值分配内存，但是在不离开作用域的情况下重复执行这个字面值只是对它的存储空间进行必要的重新初始化。这种重复执行只可能发生为带有goto语句的循环中，因为在任何迭代语句中，复合字面值将位于循环体的作用域中，在每次迭代时都将重新进入作用域。

例子

下面这个循环用指向一个数组的指针填充ptrs，并且*(ptrs[i]) == 4。

```
int * ptrs[5]; int i = 0;
again:
   ptrs[i] = (int [1]){i};
   if (++i<5) goto again;
```

下面的代码用指向不同数组的指针填充ptrs，并且*(ptrs[i]) == i。

```
int * ptrs[5]; int i = 0;
ptrs[i] = (int [1]){i++}; }
ptrs[i] = (int [1]){i++}; }
ptrs[i] = (int [1]){i++}; }
ptrs[i] = (int [1]){i++}; }
ptrs[i] = (int [1]){i++}; }
```

下面这个循环用未定义的指针（野指针）填充ptrs，因为每个字面值数组在循环的每次迭代结束时被销毁。

```
int *ptrs[5];
for(int i=0; i<5; i++) { ptrs[i] = (int [1]){i}; }
```

参考：初始化值 第4.6节；可变长度的数组 第5.4.5节。

7.5 单目表达式

在接下来的各节中，将讨论几种类型的单目表达式。

类型转换表达式：

单目表达式
(类型名称) 类型转换表达式

单目表达式：

后缀表达式
sizeof表达式
单目负号表达式
单目正号表达式
逻辑负表达式
位反表达式

地址表达式
间接访问表达式
前缀增值表达式
前缀减值表达式

单目操作符的优先级低于后缀表达式，但高于所有的双目和三目操作符。例如，表达式*x++被解释为*(x++)，而不是(*x)++。

参考：双目表达式 第7.6节；后缀表达式 第7.4节；优先级 第7.2.1节；单目正号操作符 第7.5.3节。

7.5.1 类型转换

类型转换表达式由左括号、类型名称、右括号和操作数表达式组成。前面已经显示了它的语法，和“单目表达式”并列。

类型转换导致操作数的值被转换为括号内的类型。任何允许进行的转换（参见第6.3.1节）都可以通过类型转换表达式进行，其结果不是左值。

例子

```
extern char *alloc();
struct S *p;
p = (struct S *) alloc(sizeof(struct S));
```

有些C编译器会不正确地忽略那些唯一的转换效果就是使一个值比正常“更窄”的类型转换。

例子

假设unsigned int类型用16位表示，unsigned类型用32位表示。则下面这个表达式的值

```
(unsigned)(unsigned short)0xFFFFFF
```

应该是0xFFFF，由于类型转换(unsigned short)应该使0xFFFFFF截短为16位，而(unsigned)这个类型转换应该把这个值加宽到32位。有缺陷的编译器无法实现这种截短效果，会生成代码不加修改地传递0xFFFFFF这个值。类似，对于下面这个表达式

```
(double)(float)3.1415926535897932384
```

有缺陷的编译器并不会产生代码，把π的近似值的精度降低到float，而是会不加修改地传递这个双精度值。

使用非标准的编译器时，为了使程序具有最大限度的可移植性，程序员应该截断值，然后再把它们存储到变量中，或者，在操作整数时，可以使用显式的掩码操作（例如使用二进制位AND操作符&），而不是依赖变窄转换。

参考：位AND操作符 第7.6.6节；类型转换 第6章；类型名称 第5.12节。

7.5.2 sizeof操作符

sizeof操作符用于获取类型或数据对象的长度：

sizeof表达式：

```
sizeof ( 类型名称 )
sizeof 单目表达式
```

sizeof表达式具有两种形式：sizeof操作符后面跟一个带括号的类型名称；sizeof操作符后面

跟一个操作数表达式。它的结果是个整数常量值，并且不是左值。在标准C中，sizeof的结果是stddef.h头文件所定义的无符号类型size_t。传统C常常使用int或long作为sizeof的结果类型。按照C的优先级规则，sizeof (long) − 2被解释为(sizeof (long) − 2))，而不是sizeof ((long) − (2))。

对加括号的类型名称使用sizeof操作符产生指定类型的对象的长度，也就是该类型的对象将占据的内存数量（按存储单位计），包括内部或缀尾的任何填充空间。按照定义，sizeof作用于任何字符类型所产生的结果是1。类型名称不能是不完整的数组类型（即没有显式地声明长度的数组）、函数类型或void类型。

sizeof操作符作用于表达式时所产生的结果与作用于该表达式的类型名称时所产生的结果相同。在确定表达式的类型时，sizeof操作符并不会对表达式执行任何寻常转换。这就允许使用sizeof确定数组的总长度，而不会把数组名转换为指针。但是，如果表达式包含了执行寻常转换的操作符，则在确定表达式的类型时就会考虑这些转换。sizeof的操作符不能是不完整的数组类型或函数类型。但是，如果sizeof的操作数是被声明为数组或函数类型的形式参数名称，它的返回值就是按照正常规则对这些类型的形参进行转换所产生的指针类型的长度。在标准C中，sizeof的操作数不能是指定了结构或联合对象中的位段的左值，但有些非标准的C编译器允许这种做法，并返回该成员所声明类型的长度（忽略位段指定）。

例子

下面是sizeof操作符的一些使用例子。假设short类型的对象占据2个字节，int类型的对象占据4个字节。

表达式	值
sizeof(char)	1
sizeof(int)	4
short s; … sizeof(s)	2
short s; … sizeof(s+0)	4（+的结果具有int类型）
short sa[10]; … sizeof(sa)	20
extern int ia[]; … sizeof(ia)	非法（类型不完整）

当sizeof作用于表达式时，这个表达式将会在编译时进行分析，以确定它的类型，但这个表达式并不会被求值。当sizeof的参数是个类型名称时，可能会声明一个类型，这是操作的副作用。

如果一个可变长度的数组类型名称出现在sizeof表达式中，并且这个数组长度的值会影响sizeof表达式的值时，则数组长度表达式总是会完全求值，包括随之产生的副作用。如果数组长度的值并不影响sizeof的结果，则数组长度表达式是否进行求值是未定义的。

例子

在下面的语句中，j并没有增值，但n增了值。函数调用f(n)可能执行，也可能未执行。这个调用并不一定执行，因为sizeof表达式只计算一个指向可变长度数组的指针的长度，而不是取决于这个数组本身的长度。

```
size_t z = sizeof(j++);
size_t x = sizeof(int [n++]);
size_t y = sizeof(int (*)[f(n)]);
```

下面这个表达式的效果

```
sizeof(struct S {int a,b;})
```

是在标准C中创建一种新类型，尽管这是一种不良的做法。这个类型以后可以在源文件中被引用（在C++中，这种做法是非法的）。

参考：数组类型 第5.4节；C++兼容性 第5.8节；size_t 第11.1节；存储单位 第6.1.1节；类型名称 第5.12节；无符号类型 第5.1.2节；寻常双目转换 第6.3.4节；可变长度数组 第5.4.5节；void类型 第5.9节。

7.5.3 单目负号和正号

单目负号操作符计算它的操作数的算术相反值。单目正号操作符（标准C所引入）简单地产生它的操作数的值：

单目负号操作符：

```
- 类型转换表达式
```

单目正号操作符：　　(C89)

```
+ 类型转换表达式
```

这两个操作符的操作数可以是任何算术类型，并执行寻常单目转换，其结果是提升后的类型，并且不再是左值。

单目负号表达式-e是0-(e)的简单记法，这两个表达式执行相同的计算。如果操作数是个有符号整数或浮点数并且发生了溢出，这种计算可能会产生不可预料的结果。对于无符号整型操作数k，其结果总是个无符号整数，等于2^n-k，其中n是表示结果的位数。由于结果是无符号数，所以它不可能是负数。这看上去比较奇怪，但是注意对于任何无符号整数x以及具有良好的-x定义的有符号整数x，(-x)+x的结果是0。

单目正号表达式+e是0+(e)的简单记法。

参考：浮点类型 第5.2节；整数类型 第5.1节；左值 第7.1节；溢出 第7.2.2节；减法操作符- 第7.6.2节；无符号类型 第5.1.2节；寻常单目转换 第6.3.3节。

7.5.4 逻辑反

单目操作符!计算它的操作数的逻辑反。它的操作数可以是任何标量类型：

逻辑反表达式：

```
! 类型转换表达式
```

操作数将会执行寻常单目转换。!操作符的结果是int类型。如果操作数为0（以及null指针或浮点数0.0），其结果为1。如果操作数不是0（或null、0.0），其结果为0。它所产生的结果是个左值。表达式!(x)与(x)==0的含义相同。

例子

```
#define assert(x,s) if (!(x)) assertion_failure(s)
…
assert(num_cases > 0, "No test cases.");
average = total_points/num_cases;
```

使用assert.h宏时预料到一个问题，即除零错误。这种错误较难检测。假设assertion_failure是个函数，它接受一个字符串，并把它作为错误信息报告给用户。标准头文件assert.h中有一个

类似的宏assert。

参考：assert 第19.1节；相等操作符== 第7.6.5节；浮点类型 第5.2节；整数类型 第5.1节；左值 第7.1节；指针类型 第5.3节；标量类型 第5章；寻常单目转换 第6.3.3节。

7.5.5 位反

单目操作符~计算它的操作数的位反（NOT）：

位反表达式：

~ 类型转换表达式

操作数将会执行寻常单目转换，操作数可以是任何整数类型。在~e的表示形式中，每个位都是（经过转换的）操作数e的对应位的反值。它的结果不是左值。

例子

如果i是个值为0xF0F0（1111000011110000_2）的16位整数，则~i的值是0x0F0F（0000111100001111_2）。

由于不同的计算机可能用不同的表示形式表示有符号整数，所以对于有符号整数应用~操作符的结果可能是不可移植的。为了实现可移植的代码，我们建议只对无符号操作数使用~操作符。对于无符号操作数e，如果e经过转换后的类型是unsigned，~e的值就是UNIT_MAX−e。如果e经过转换后的类型是unsigned long，~e的值就是ULONG_MAX−e。UNIT_MAX和ULONG_MAX的值是在标准头文件limits.h中定义的。

参考：整数类型 第5.1节；limits.h 第5.1.1节；左值 第7.1节；有符号类型 第5.1.1节；无符号类型 第5.1.2节；寻常单目转换 第6.3.3节。

7.5.6 取地址操作符

单目操作符&返回指向它的操作数的指针：

取地址表达式：

& 类型转换表达式

&的操作数必须是函数指示符或指定了某个对象的左值。如果它是个左值，这个对象就无法用存储类别register声明，也不能被声明为位段。如果&的操作数的类型是“T”，则结果类型是“指向T的指针”。&的操作数并不会执行寻常转换，它的结果不会是左值。

作用于函数指示符的取地址操作符产生一个指向函数的指针。由于函数指示符可以通过寻常转换规则转换为函数指针，所以极少需要对函数使用&操作符。事实上，有些标准C之前的编译器禁止这种用法。

例子

```
extern int f();
int (*fp)();
…
fp = &f;    /* OK; & yields a pointer to f */
fp = f;     /* OK; usual conversions yield a pointer to f */
```

取地址操作符所产生的函数指针在C程序的执行期间一直是有效的。对于取地址操作符所产生的对象指针，只要为这个对象所分配的存储空间一直有效，这个对象指针也一直有效。如果&

的操作数是个具有静态范围的变量的左值，它所产生的指针在程序的整个执行期间一直有效。如果操作数是个自动变量，当包含这个变量声明的代码块处于活动时，这个指针总是有效的。如果操作数是个动态分配的对象（例如通过malloc分配），在这块内存被显式释放之前，这个指针总是有效的。

在标准C中，取地址操作符的效果在一个方面与传统C不同。在标准C中，当取地址操作符作用于“T数组”类型的左值时，它产生一个“指向T类型数组的指针”类型的值。而许多非标准的C编译器把&a看成与a相同，也就是指向a的第1个元素的指针。这两种解释相互不一致，但标准C所采用的规则与&的解释更为一致。

例子

在下面这个标准C程序片段中，对p的所有赋值都是等价的，对i的所有赋值也是等价的：

```
int a[10], *p, i;
…
p = &a[0];  p = a;   p = *&a;
i = a[0];   i = *a;  i = **&a;
```

参考：数组类型 第5.4节；函数指示符 第7.1节；函数类型 第5.8节；左值 第7.1节；指针类型 第5.3节；register存储类别 第4.3节。

7.5.7 间接访问

单目操作符*通过指针执行间接访问。&和*操作符执行相反的操作：如果x是个变量，表达式*&x与x相同。

间接访问表达式：

* 类型转换表达式

操作数必须是个指针。对于某种可能的限定类型T，如果操作数的类型是“指向T的指针”，则结果的类型是简单的“T”（具有相同的限定形式）。如果指针指向一个对象，则结果是个引用该对象的左值。如果指针指向一个函数，则结果是个函数指示符。

例子

```
int i,*p;
const int *pc;
...
p   = &i;    /* p now points to variable i */
*p  = 10; /* sets value of i to 10 */
pc  = &i; /* pc now points to i, too */
*pc = 10;    /* invalid, *pc has type 'const int' */
```

间接访问操作符的操作数将会执行寻常单目转换。唯一相关的转换是从数组和函数指示符转换为指针。因此，如果f是个函数指示符，则表达式*&f和*f是相同的。在后面这种情况下，f通过寻常转换被转换为&f。

对非法指针或null指针应用*操作符的结果是未定义的。在有些编译器中，解引用null指针将导致程序终止。在另一些编译器中，null指针相当于指向一块内容不可预测的内存。

参考：数组类型 第5.4节；函数指示符 第7.1节；函数类型 第5.8节；左值 第7.1节；指针类型 第5.3节；寻常单目转换 第6.3.3节。

7.5.8 前缀增值和减值操作符

单目操作符++和--分别用于把它们的操作数的值增加1或减少1，同时把操作数修改后的值作为结果。它们是具有副作用的操作（这两个操作符还存在后缀形式）。

前缀增值表达式：

++ 单目表达式

前缀减值表达式：

-- 单目表达式

这两个操作符的操作数都必须是可修改的左值，并且可以是任何实数算术类型或指针类型。如果是++，操作数的值就增加常量1。如果是--，操作数的值就减少常量1。不管是哪种情况，结果被保存到左值中，并作为操作数的新值。操作的结果不再是左值。在执行加法和减法之前，将会对操作数以及常量1进行寻常双目转换。在存储新值时，将会执行寻常赋值转换。结果的类型是左值在转换之前的类型。

如果操作数是个指针，例如是“指向T的指针（T是某种类型）”，++操作符的效果相当于把指针从它所指向的对象前移一步，就像指向T类型的数组中的下一个元素（在字节编址的计算机中，这意味着把指针向前移动sizeof(T)个字节）。--操作符的效果相当于使指针指向T类型数组中的前一个元素。

例子

下面这个strrev函数把它的第1个参数的一份反序拷贝复制到第2个参数：

```
int strrev( const char *s1, char *s2 )
{
    const char *p = s1;
    while (*p++); /* Locate end of first string. */
    --p;          /* Overshot: back up to the null. */
    /* Now copy the characters in reverse order. */
    while (p > s1)
        *s2++ = *--p;
    *s2 = '\0';  /* Terminate the result string. */
}
```

如果发生了溢出并且操作数是有符号整数类型或浮点数类型，这些操作的结果是不可预测的。对无符号类型的最大可表示的值进行增值的结果是0。对无符号类型的0进行减值的结果是该无符号类型的最大可表示值。

表达式++e和e += 1相同，--e与e -= 1相同。当增值和减值操作符所产生的值未被使用时，前缀形式和后缀形式具有相同的效果。也就是说，e++;这条语句的效果与++e;相同，e--;这条语句的效果与--e;相同。

参考：加法 第7.6.2节；数组类型 第5.4节；赋值转换 第6.3.2节；复合赋值 第7.9.2节；表达式语句 第8.2节；浮点类型 第5.2节；整数类型 第5.1节；左值 第7.1节；溢出 第7.2.2节；指针类型 第5.3节；后缀增值和减值表达式 第7.4.4节；标量类型 第5章；有符号类型 第5.1.1节；减法 第7.6.2节；无符号类型 第5.1.2节；寻常双目转换 第6.3.4节。

7.6 双目操作符表达式

双目操作符表达式由两个操作数以及中间的一个双目操作符组成。这里的“双目”这个术语简单地表示有两个操作数，它与二进制数字系统并无任何关系。表7-4列出了双目表达式的类型以及它们的操作数的类型，以递减的优先级为序。所有这些操作符都具有左结合性。

表7-4 双目操作符表达式

表达式的类型	操作符	操作数	结果
乘除表达式	* /	算术	算术
	%	整数	整数
加减表达式	+	算术	算术
		指针 + 整数 或整数 + 指针	指针
	-	算术	算术
		指针 - 整数	算术
		指针 - 指针	整数
移位表达式	<< >>	整数	整数
关系表达式	< <= >= >	算术或指针	0或1
相等表达式	== !=	算术或指针	0或1
位与表达式	&	整数	整数
位异或表达式	^	整数	整数
位或表达式	\|	整数	整数

对于本节所描述的每个双目操作符，在执行操作之前，两个操作数都会进行完整的求值(但并不按照某个特定的顺序)。

参考：求值的顺序 第7.2.2节；优先级 第7.2.1节。

7.6.1 乘除操作符

三个乘除操作符*（乘法）、/（除法）和%（求余）具有相同的优先级，并且都具有左结合性：

乘除表达式：

类型转换表达式
乘除表达式 乘除操作符 类型转换表达式

乘除操作符：下列之一

* / %

参考：优先级 第7.2.1节。

乘法 双目操作符*表示乘法。每个操作数可以是任何算术类型。操作数将会执行寻常双目转换，结果的类型是操作数经过转换的类型。乘法运算的结果不再是左值。对于整型操作数，所执行的是整数乘法。对于浮点型操作数，所执行的是浮点数乘法。

如果发生了溢出并且操作数是有符号整数类型或浮点类型（经过转换之后），乘法操作符可能会产生不可预料的结果。如果操作数是无符号整数，结果就是同余模数2^n减去真实数学运算结果的值（其中n是表示无符号运算结果的位数）。

参考：算术类型 第5章；浮点类型 第5.2节；整数类型 第5.1节；左值 第7.1节；求值的顺序

第7.12节；溢出 第7.2.2节；有符号类型 第5.1.1节；无符号类型 第5.1.2节；寻常双目转换 第6.3.4节。

除法 双目操作符/表示除法。每个操作数可以是任何算术类型。操作数将会执行寻常双目转换，结果的类型是操作数经过转换的类型。除法运算的结果不再是左值。

对于浮点型操作数，所执行的是浮点除法。对于整型操作数，如果操作数的算术商不是整数，小数部分就被丢弃（舍去为0）。在C99之前，如果有任一操作数为负数，C编译器可以选择将小数部分舍去为0或入位为1。库函数div和ldiv对操作数为负的除法进行了良好的定义。

如果发生了溢出并且操作数是有符号整数类型或浮点类型（经过转换之后），除法操作符可能会产生不可预料的结果。注意，对于用补码表示的有符号整数，如果最大可表示的负整数除以－1，就会发生溢出。它的数学结果是个无法表示的正整数。如果两个操作数都是无符号类型，就不可能发生溢出。

除零错误（整数或浮点数）的结果是未定义的。

参考：算术类型 第5章；div 第17.1节；浮点类型 第5.2节；整数类型 第5.1节；ldiv 第17.1节；左值 第7.1节；溢出 第7.2.2节；有符号类型 第5.1.1节；无符号类型 第5.1.2节；寻常转换 第6.3.4节。

余数 双目操作符%计算第1个操作数除以第2个操作数所产生的余数。每个操作数可以是任何整数类型。操作数会执行寻常双目转换，结果的类型是操作数经过转换的类型。运算结果不再是左值。库函数div、ldiv和fmod也用于计算整数和浮点数的余数。

如果a/b是可以表示的，(a/b) * b + a % b总是等于a，因此余数计算的行为是伴生于整数除法的。如前一节所述，在C99之前，当其中一个操作数为负时，除法操作符的行为因编译器而异。这使余数操作符同样依赖于编译器。

例子

下面这个gcd函数根据Euclid算法计算最大公约数。它的结果是能够被x和y整除的最大整数：

```
unsigned gcd(unsigned x, unsigned y)
{
    while ( y != 0 ) {
        unsigned temp = y;
        y = x % y;
        x = temp;
    }
    return x;
}
```

如果余数操作符作用于两个操作数时将产生溢出，将会出现意想不到的效果。如果有符号整数是用补码表示的，最大可表示的负整数除以－1将会发生溢出。这个除法的数学结果是个无法表示的正整数，因此这个结果是不可预料的，尽管余数（0）是可以表示的。如果操作数是无符号整数，就不可能发生溢出。

如果第2个操作数为0，余数操作符的结果是未定义的。

参考：div、ldiv 第17.1节；fmod 第17.3节；整数类型 第5.1节；左值 第7.1节；溢出 第7.2.2节；有符号类型 第5.1.1节；无符号类型 第5.1.2节；寻常双目转换 第6.3.4节。

7.6.2 加减操作符

两个加减操作符+（加法）和－（减法）具有相同的优先级，并且都具有左结合性：

加减表达式：

乘除表达式
乘除表达式　加减操作符　乘除表达式

加法操作符：下列之一

+　-

加法 双目操作符+表示加法。操作数将会执行寻常双目转换。操作数可以都是算术类型，也可以一个为对象指针另一个为整数。它不允许使用其他类型的操作数。加法运算的结果不再是左值。

当操作数都是算术类型时，结果类型就是转换之后的操作数类型。对于整型操作数，所执行的是整型加法。对于浮点型操作数，所执行的是浮点型加法。

当指针p与整数k相加时，它假定p所指向的对象位于一个该对象类型的数组之内，或者是数组最后一个对象之后的那个位置。这种加法的结果是个指针，指向这个假想数组中（或数组尾部之后的那个位置）p原先所指向对象向后k个位置的对象。例如，p + 1所指向的对象就是p原先所指向对象之后的那个对象，p + (－1)指向p原先所指向对象之前的那个对象。如果指针p或p + k并不位于这个数组内部（或数组之后的那个位置），则这种加法的行为是未定义的。如果p是个函数指针或是void *类型的指针，这种加法也是非法的。

例子

假设有一台采用字节编址的计算机，并且int的长度为4个字节。假设a是个包含10个整数的数组，它从地址0x100000开始。假设ip是个指向整数的指针，并把数组a的第1个元素的地址赋值给它。最后，假设i是当前值为6的整型变量。上面这些情况可以用下面的代码来表示：

```
int *ip, i, a[10];
ip = &a[0];
i = 6;
```

ip + i的类型是什么？由于整数的长度是4个字节，所以表达式ip + i的结果是0x100000 + 4 * 6，或0x100018（十进制的24等于十六进制的18）。

例子

指向多维数组或可变长度数组的指针采用类似的操作：

```
int n = 3; int m = 5;
double rect[n][m];
double (*p)[m];
p = rect; /* same as p = &rect[0]; */
p++; /* now p == &rect[1] */
```

标识符p指向一个double[m]类型的对象，这是一个包含5个双精度浮点数的数组，也就是矩阵rect的一行。表达式p++使p指向rect的下一行，使它前进5 * sizeof(double)个存储单位。

如果发生了溢出并且操作数（经过转换之后）是有符号整数或浮点数，或者其中一个操作数是指针，加法操作符可能会产生出人意料的结果。如果两个操作数都是无符号整数，则结果是同余模数2^n减去运算的真实数学结果之值（其中n是用于表示无符号结果的位数）。

参考：数组类型 第5.4节；浮点类型 第5.2节；整数类型 第5.1节；左值 第7.1节；多维数组 第5.4.2节；求值的顺序 第7.12节；溢出 第7.2.2节；指针表示形式 第5.3.2节；指针类型 第5.3节；标量类型 第5章；有符号类型 第5.1.1节；无符号类型 第5.1.2节；寻常双目转换 第6.3.4节；可变长度的数组 第5.4.5节。

减法 双目操作符-表示减法。操作数将会执行寻常双目转换。操作数可以都是算术类型或者都是指向兼容对象类型的指针（忽略任何类型限定符），或者左操作数是个指针，右操作数是个整数。减法运算的结果不是左值。

如果操作数都是算术类型，结果类型就是转换之后的操作数类型。对于整型操作数，所执行的是整型减法。对于浮点型操作数，所执行的是浮点型减法。

例子

无符号整数减去无符号整数的结果总是另一个无符号整数，因此不可能是负值。但是，无符号整数总是遵守下面的规则：

```
(a+(b-a)) == b
```

和

```
(a-(a-b)) == b
```

指针减去整数与指针和整数的相加类似。当指针p减去整数k时，它假定p所指向的对象位于一个该对象类型的数组之内，或者是数组最后一个对象之后的那个位置。减法的结果是个指针，指向这个假想数组中（或数组尾部之后的那个位置）p原先所指向对象向后－k个位置的对象。例如，p－1所指向的对象就是p原先所指向对象之前的那个对象，p－(－1)指向p原先所指向对象之后的那个对象。如果指针p或p－k并不位于这个数组的内部（或数组之后的那个位置），则这种减法的行为是未定义的。如果p是个函数指针或是void *类型的指针，这种减法也是非法的。

假设有两个相同类型的指针p和q，p－q的差是整数k，q加上k的结果是p。指数相减之差的类型是stddef.h头文件所定义的有符号整数类型ptrdiff_t（在标准C之前，这个类型可以是int或long，取决于编译器）。只有当两个指针指向同一个数组中的对象（或数组之后的那个位置），它们相减的结果才具有良好的定义，并且是可以移植的。差k是它们所指向的两个对象的下标之差。如果指针p或p － q位于数组之外，指针减法的行为就是未定义的。如果p或q是函数指针，或者void *类型的指针，它们的减法是非法的。

如果发生了溢出并且操作数（在转换之后）是有符号整数或浮点类型，或者任一操作数为指针类型，减法运算可能会产生出人意料的结果。如果两个操作数都是无符号整数，则结果是同余模数2^n减去运算的真实数学结果之差（其中n是用于表示无符号结果的位数）。

参考：数组类型 第5.4节；浮点类型 第5.2节；整数类型 第5.1节；左值 第7.1节；溢出 第7.2.2节；指针表示形式 第5.3.2节；指针类型 第5.3节；ptrdiff_t 第11.1节；标量类型 第5章；有符号类型 第5.1.1节；类型兼容性 第5.11节；类型限定符 第4.4.3节；无符号类型 第5.1.2节；寻常双目转换 第6.3.4节。

7.6.3 移位操作符

双目操作符<<表示向左移位，双目操作符>>表示向右移位。这两个操作符具有相同的优先级，并都具有左结合性：

移位表达式：

加减表达式
移位表达式 移位操作符 加减表达式

移位操作符：下列之一

<< >>

每个操作数必须是整数类型。每个操作数将分别执行寻常单目转换，结果的类型是经过转换之后的左操作数的类型（在标准C之前的编译器中，将对两个操作数执行寻常双目转换）。移位的结果不是左值。

第1个操作数表示被移位的对象，第2个操作符指定了对第1个操作数进行移位的位数。移位操作的方向是由实际所使用的移位操作符决定的。<<操作符对左操作数进行向左移位，左边被移掉的位将被丢弃，右边空出来的位置用0填充。>>操作符对左操作数进行向右移位，右边被移掉的位将被丢弃，左边空出来的位置所填充的值取决于左操作数的类型。如果左操作数是个无符号值（或者是非负的有符号值），左边空出来的位用0填充。但是，如果左操作数为负的有符号值，则左边空出来的位用0还是用左操作数最左边的位填充取决于编译器。因此，当左操作数是负的有符号值并且右操作数不为零时，移位操作符>>的结果是不可移植的。

如果右操作数是个负值，移位操作符的结果是未定义的。因此，指定一个负的移位距离并不保证<<向右移位或者>>向左移位。如果右操作数的值大于或等于经过转换之后的左操作数的宽度（以位为单位），移位操作的结果也是未定义的。右操作数可以是0，在这种情况下，并不会发生移位，结果值等于经过转换之后的左操作数。

例子

我们可以编写下面这样的表达式探索操作符的优先级和结合性。这种写法看上去比较悦目，但在语义上容易让人混淆：

```
b << 4 >> 8
```

如果b是个16位的无符号值，则这个表达式抽取中间的8位。和往常一样，在可能令人混淆的地方，最好使用括号使它变得清晰：

```
(b << 4) >> 8
```

例子

下面这个例子使用无符号移位操作，通过二进制算法计算两个整数的最大公约数。这个方法比Euclidean算法更为复杂，但它的速度更快，因为在有些C编译器中，余数运算的速度较慢，尤其是当操作数为无符号整数时。

```
unsigned binary_gcd(unsigned x, unsigned y)
{
    unsigned temp;
    unsigned common_power_of_two = 0;
    if (x == 0) return y; /* Special cases */
    if (y == 0) return x;

    /* Find the largest power of two
       that divides both x and y. */
    while (((x | y) & 1) == 0) {
        x = x >> 1; /* or: "x >>= 1;" */
```

```
        y = y >> 1;
        ++common_power_of_two;
    }
    while ((x & 1) == 0) x = x >> 1;

    while (y) {
        /* x is odd and y is nonzero here. */
        while ((y & 1) == 0) y = y >> 1;
        /* x and y are odd here. */
        temp = y;
        if (x > y) y = x - y;
        else y = y - x;
        x = temp;
        /* Now x has the old value of y, which is odd.
           y is even, because it is the difference of two odd
           numbers; therefore it will be right-shifted
           at least once on the next iteration. */
    }
    return (x << common_power_of_two);
}
```

参考：整数类型 第5.1节；左值 第7.1节；优先级 第7.2.1节；有符号类型 第5.1.1节；寻常单目转换 第6.3.3节。

7.6.4 关系操作符

双目操作符<、<=、>和>=用于对它们的操作数进行比较：

关系表达式：

移位表达式
关系表达式 关系操作符 移位表达式

关系操作符：下列之一

< <= > >=

操作数将会执行寻常双目转换。操作数可以都是实数（非复数）算术类型，或者都是指向兼容类型的指针，或者都是指向兼容的不完整类型的指针。如果指针类型存在任何类型限定符，并不会影响比较的结果。比较的结果总是int类型，它的值为0或1。比较结果不是左值。

操作符<测试“小于”这个关系，操作符<=测试“小于或等于”这个关系，操作符>测试“大于”这个关系，操作符>=测试“大于或等于”这个关系。如果两个操作数具有指定的关系，比较结果就是1，否则为0。

在标准C中，浮点类型运算的实现可能包含像NaN这样的值，这种值是无序的。在关系表达式中使用这种值可能会导致“非法操作”异常，并且关系操作的值将为假。第17.16节讨论了一些函数，它们在这种情况下比内置的操作符具有更合理的行为。

对于整型操作数，关系操作符所执行的是整型比较（有符号或无符号，依合适而定）。对于浮点型操作数，关系操作符所执行的是浮点型比较。对于指针操作数，比较的结果取决于两个指针所指向的对象在地址空间中的相对位置。只有当两个指针所指向的对象位于同一个数组或结构中时，比较的结果才是有意义的。在这种情况下，“大于”意味着“在数组中具有更高的索引”或者“在结构成员声明列表中出现得较晚”。数组的一种特殊情况，也就是指向数组尾部之

后的那个对象的指针，也具有良好的定义，可以与指向数组内的对象的指针进行比较。指向同一个联合对象的成员的所有指针的比较结果都是相同的。

例子

可以编写像3<x<7这样的表达式，但是，它的含义和正常的数学记法并不相同。由于操作符的左结合性，它被解释为“(3<x)<7”。由于(3<x)的结果是0或1，它们都小于7，所以3<x<7的结果总是1。为了表达正常数学记法的含义，可以使用逻辑AND操作符，例如3<x && x<7。

例子

对混合类型使用关系操作符时，应该小心谨慎。下面是一个特别容易混淆的例子：

```
-1 < (unsigned) 0
```

读者可能会认为这个表达式的结果总是1（真），因为－1总是小于0。但是，在比较之前，寻常双目转换会把－1转换为一个（很大的）无符号整数。而无符号整数不可能小于0。因此，这个表达式的结果总是0（假）。

有些旧式的编译器允许指针和整数之间的关系比较，但这种比较实际上是被禁止的。这些旧式的编译器可能会把这类比较当成有符号整数或无符号整数的比较。

参考：算术类型 第5章；数组类型 第5.4节；位AND操作符& 第7.6.6节；兼容类型 第5.11节；浮点类型 第5.2节；不完整类型 第5.4节、第5.6.1节；整数类型 第5.1节；逻辑AND操作符&& 第7.7节；左值 第7.1节；NaN 第5.2节；指针类型 第5.3节；优先级 第 7.2.1节；有符号类型 第5.1.1节；类型限定符 第4.4.3节；无符号类型 第5.1.2节；寻常双目转换 第6.3.4节。

7.6.5 相等操作符

双目操作符==和!=用于比较操作数的相等性：

相等表达式：

```
关系表达式
相等表达式   相等操作符   关系表达式
```

相等操作符：下列之一

```
==   !=
```

它们允许使用下面这些类型的操作数：

1. 两个操作数可以都是算术类型，包括复数。

2. 两个操作数可以是指向兼容类型的指针，或者都是void *类型的指针。

3. 一个操作数可以是个指向对象或不完整类型的指针，另一个可以是void *类型的指针。第1个操作数将被转换为void *类型。

4. 一个操作数可以是指针，另一个操作数可以是null指针常量（整数值0）。

如果操作数是指针，它们所指向的类型是否具有类型限定符并不会影响是否允许比较，也不影响比较的结果。算术类型的操作数将会执行寻常双目转换。比较的结果总是int类型，其值为0或1。比较的结果不是左值。

对于整型操作数，相等操作符所执行的是整型比较。对于浮点型操作数，相等操作符所执行的是浮点型比较。只有当两个指针操作数满足下面的条件时，它们的比较结果才是相等的：

1. 两个指针指向同一个对象或函数。

2. 两个指针都是null指针。

3. 两个指针都指向同一个数组对象的最后一个元素之后的那个对象。

操作符==测试“等于”，操作符!=测试“不等于”。如果两个操作数具有指定的关系，比较结果就是1，否则为0。

对于复数类型的操作数（C99），两个操作数的实部和虚部都必须进行比较，只有它们比较的结果都相等，这两个操作数才是相等的。如果一个操作数是实数类型，另一个操作数是复数类型，则在比较之前首先把实数操作数转换为复数类型。两个操作数所执行的寻常双目转换将使它们具有相同的精度。

结构和联合无法比较相等性，尽管可以对它们进行赋值。由于对齐限制，结构或联合内部的填充空间可能包含任意的值，为了比较它们的相等性或者执行对它们进行修改的所有操作，将要付出不可接受的开销。

双目相等操作符具有相同的优先级（但低于<、<=、>和>=），并且都具有左结合性。

例子

表达式x==y==7的含义与正常的数学记法不同。由于左结合性，它被解释为(x==y)==7。由于(x==y)的结果是0或1，它们都不等于7，所以x==y==7的结果总是0。为了表达正常数学记法的含义，可以使用逻辑AND操作符，如下所示：

```
x==y && y==7
```

例子

除了位AND和位OR操作符之外，还有一个位XOR操作符。但是，在逻辑AND和逻辑OR操作符之外，不存在逻辑XOR操作符。操作符!=可以实现逻辑XOR操作符的目的：程序员可以编写像a<b != c<d这样的表达式，只有当a<b和c<d有且只有1个结果为1时，这个表达式的值才是1，否则为0。如果任一操作数的值为0和1之外的值，则可以对两个操作数使用单目操作符!：对于!x != !y，当且只当x和y有一个为非零时，这个表达式的值为1，否则为0。按照类似的方式，操作符==可以作为逻辑等价操作符（EQV）使用。

例子

一个常见的C编程错误是在需要使用==（比较）操作符时误用了=（赋值）操作符。其他一些编程语言使用=测试相等性。由于风格的原因，如果需要测试一个赋值表达式的值是否为0，最好显式地编写“!=0”，以明确意图。例如，程序的阅读者不太清楚下面这个循环是正确的还是出现了打字错误：

```
while (x = next_item()) {
/* Should this be "x==next_item()" ?? */
    …
}
```

如果原来的形式是正确的，则可以使用下面的代码使它的意图更为清晰：

```
while ((x = next_item()) != 0) {
    …
}
```

参考：对齐限制 第5.6.4节、第6.1.3节；位操作符 第7.6.6节；兼容类型 第5.11节；逻辑操作符 第7.5.4节、第7.7节；左值 第7.1节；空指针 第5.3.2节；指针类型 第5.3节；优先级 第7.2.1

节；赋值操作符= 第7.9.1节；类型限定符 第4.4.3节；寻常双目转换 第6.3.4节；void * 第5.3.1节。

7.6.6 位操作符

双目操作符&、^ 和 | 分别表示“位与”、“位异或”和“位或”功能。它们都具有左结合性，但它们的不同优先级决定了表达式的求值顺序。它们的操作数必须都是整数类型，并且将执行寻常双目转换。位操作的结果不是左值。

位或表达式：

位异或表达式
位或表达式 | 位异或表达式

位异或表达式：

位与表达式
位异或表达式 ^ 位与表达式

位与表达式：

相等表达式
位与表达式 & 相等表达式

这些操作符所产生的结果的每个位等于两个（转换后的）操作数对应的位的一个布尔函数的结果：

- 如果两个参数位都是1，&（与）函数所产生的结果是1，否则为0。
- 如果一个参数位为1，另一个参数位为0，^（异或）函数所产生的结果是1。如果两个参数位都是0或1，这个函数所产生的结果是0。
- 如果有任一参数位为1，|（或）函数所产生的结果是1，否则为0。

下面对这些行为进行了总结：

a	b	a & b	a ^ b	a \| b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

每个位操作符都满足交换律，并且具有相同的结合性。编译器允许对包含符合第7.12节所描述的限制的操作符的表达式进行重新排列。

为了实现可移植的代码，我们推荐只对无符号操作数使用位操作符。对于大多数使用补码表示有符号整数的计算机，使用有符号操作数也不会出现问题。但是，在其他计算机上，这种操作可能会失败。

程序员必须小心，不要在应该使用逻辑AND和OR操作符（&&和||）的地方误用位操作符&和|。只有当参数不具有副作用并且是布尔类型（0或1）时，位操作符与对应的逻辑操作符的结果才是相同的。而且，按位操作符总是对它们的操作数求值，因此，如果左操作数的值足够用来决定最终的表达式结果，那么逻辑操作符不对右边操作数求值。

例子

如果a是2，b是4，则a & b为0（假），a && b为1（真）。

7.6.7 整数集合的例子

接下来的几页分别显示了“整数集合”程序包的使用、声明和定义。它使用位操作符把这些集合实现为位向量。这个例子包含了一个示例程序（testset.c）、测试程序的输出、程序包头文件（set.h）以及程序包中各个函数的实现（set.c）。

参考：整数类型 第5.1节；逻辑操作符&&和|| 第7.7节；左值 第7.1节；求值的顺序 第7.12节；关系操作符 第7.6.4节；有符号类型 第5.1.1节；无符号类型 第5.1.2节；寻常单目转换 第6.3.4节。

```
#include "set.h"
int main(void)
{
    print_k_of_n(0, 4);
    print_k_of_n(1, 4);
    print_k_of_n(2, 4);
    print_k_of_n(3, 4);
    print_k_of_n(4, 4);
    print_k_of_n(3, 5);
    print_k_of_n(3, 6);
    return 0;
}
```

SET程序包的示例用法：testset.c文件

```
All the size-0 subsets of {0, 1, 2, 3}:
{}
The total number of such subsets is 1.

All the size-1 subsets of {0, 1, 2, 3}:
{0}  {1}  {2}  {3}
The total number of such subsets is 4.

All the size-2 subsets of {0, 1, 2, 3}:
{0, 1}  {0, 2}  {1, 2}  {0, 3}  {1, 3}  {2, 3}
The total number of such subsets is 6.

All the size-3 subsets of {0, 1, 2, 3}:
{0, 1, 2}  {0, 1, 3}  {0, 2, 3}  {1, 2, 3}
The total number of such subsets is 4.

All the size-4 subsets of {0, 1, 2, 3}:
{0, 1, 2, 3}
The total number of such subsets is 1.

All the size-3 subsets of {0, 1, 2, 3, 4}:
{0, 1, 2}  {0, 1, 3}  {0, 2, 3}  {1, 2, 3}
{0, 1, 4}  {0, 2, 4}  {1, 2, 4}  {0, 3, 4}
{1, 3, 4}  {2, 3, 4}
The total number of such subsets is 10.

All the size-3 subsets of {0, 1, 2, 3, 4, 5}:
{0, 1, 2}  {0, 1, 3}  {0, 2, 3}  {1, 2, 3}
{0, 1, 4}  {0, 2, 4}  {1, 2, 4}  {0, 3, 4}
{1, 3, 4}  {2, 3, 4}  {0, 1, 5}  {0, 2, 5}
{1, 2, 5}  {0, 3, 5}  {1, 3, 5}  {2, 3, 5}
{0, 4, 5}  {1, 4, 5}  {2, 4, 5}  {3, 4, 5}

The total number of such subsets is 20.
```

SET程序包：testset.c文件

```
/* set.h
A set package, suitable for sets of small integers in the
range 0 to N-1, where N is the number of bits in an unsigned
int type. Each integer is represented by a bit position; bit
i is 1 if and only if i is in the set. The low-order bit is
bit 0. */

#include <limits.h> /* defines CHAR_BIT */

/* Type SET is used to represent sets. */

typedef unsigned int SET;

/* SET_BITS: Maximum bits per set. */

#define SET_BITS              (sizeof(SET)*CHAR_BIT)

/* check(i): True if i can be a set element. */

#define check(i)            (((unsigned) (i)) < SET_BITS)

/* emptyset: A set with no elements. */

#define emptyset ((SET) 0)

/* add(s,i): Add a single integer to a set. */

#define add(set,i)          ((set) | singleset (i))

/* singleset(i): Return a set with one element in it. */

#define singleset(i)(((SET) 1) << (i))

/* intersect: Return intersection of two sets. */

#define intersect(set1,set2 )((set1) & (set2))

/* union: Return the union of two sets. */

#define union(set1,set2     )((set1) | (set2))

/* setdiff: Return a set of those elements in set1 or set2,
   but not both. */

#define setdiff(set1,set2   )((set1) ^ (set2))

/* element: True if i is in set. */

#define element(i,set       )(singleset((i)) & (set))
```

SET程序包：set.h文件（第1部分）

```
/* forallelements:  Perform the following statement once for
   every element of the set s, with the variable j set to
   that element. To print all the elements in s, just write
   int j;
   forallelements(j, s)
      printf("%d ", j);
*/

#define forallelements(j,s) \
    for ((j)=0; (j)<SET_BITS; ++(j)) if (element((j),(s)))

/* first_set_of_n_elements(n): Produce a set of size n whose
   elements are the integers from 0 through n-1.  This
   exploits the properties of unsigned subtractions. */

#define first_set_of_n_elements(n)(SET)((1<<(n))-1)

/* next_set_of_n_elements(s): Given a set of n elements,
```

```
   produce a new set of n elements.  If you start with the
   result of first_set_of_n_elements(k), and then at each
   step apply next_set_of_n_elements to the previous result,
   and keep going until a set is obtained containing m as a
   member, you will have obtained a set representing all
   possible ways of choosing k things from m things. */

extern SET next_set_of_n_elements PARM (SET x);

/* printset(s): Print a set in the form "{1, 2, 3, 4}". */

extern void printset(SET z);

/* cardinality(s): Return the number of elements in s. */

extern int cardinality(SET x);

/* print_k_of_n(k,n): Print all the sets of size k having
   elements less than n.  Try to print as many as will fit
   on each line of the output.  Also print the total number
   of such sets; it should equal n!/(k!(n-k)!)
   where n! = 1*2*…*n. */

extern void print_k_of_n(int k, int n);
```

SET程序包：set.h文件（第2部分）

```
#include <stdio.h>
#include "set.h"

int cardinality(SET x)
{

/* The following loop body is executed once for every 1-bit
   in the set x.  Each iteration, the smallest remaining
   element is removed and counted. The expression (x & -x)
   is a set containing only the smallest element in x, in
   twos-complement arithmetic. */

   int count = 0;
   while (x != emptyset) {
      x ^= (x & -x);
      ++count;
   }
   return count;
}

SET next_set_of_n_elements(SET x)
{

/* This code exploits many unusual properties of unsigned
   arithmetic.  As an illustration:
     if x == 001011001111000, then
     smallest           == 000000000001000
     ripple             == 001011010000000
     new_smallest       == 000000010000000
     ones               == 000000000000111
     the returned value == 001011010000111
   The overall idea is that you find the rightmost
   contiguous group of 1-bits. Of that group, you slide the
   leftmost 1-bit to the left one place, and slide all the
   others back to the extreme right.
   (This code was adapted from HAKMEM.) */
```

```
    SET smallest, ripple, new_smallest, ones;
    if (x == emptyset) return x;
    smallest     = (x & -x);
    ripple       = x + smallest;
    new_smallest = (ripple & -ripple);
    ones         = ((new_smallest / smallest) >> 1) - 1;
    return (ripple | ones);
}
```

SET程序包：set.c文件（第1部分）

```
void printset(SET z)
{
    int first = 1;
    int e;
    forallelements(e, z) {
        if (first) printf("{");
        else printf(", ");
        printf("%d", e);
        first = 0;
    }
    if (first) printf("{"); /* Take care of emptyset */
    printf("}");            /* Trailing punctuation */
}


#define LINE_WIDTH 54

void print_k_of_n(int k, int n)
{
    int count = 0;
    int printed_set_width = k * ((n > 10) ? 4 : 3) + 3;
    int sets_per_line = LINE_WIDTH / printed_set_width;
    SET z = first_set_of_n_elements(k);
    printf("\nAll the size-%d subsets of ", k);
    printset (first_set_of_n_elements(n));
    printf(":\n");
    do {      /* Enumerate all the sets. */
        printset(z);
        if ((++count) % sets_per_line) printf (" ");
        else printf("\n");
        z = next_set_of_n_elements(z);
    }while ((z != emptyset) && !element(n, z));
    if ((count) % sets_per_line) printf ("\n");
    printf("The total number of such subsets is %d.\n",
        count);
}
```

SET程序包：set.c文件（第2部分）

7.7 逻辑操作符表达式

逻辑操作符表达式由两个表达式以及中间的逻辑操作符（&&或||）组成。这些操作符在其他语言中有时称为“条件AND”和“条件OR”，因为如果它们的第1个表达式足以确定最终表达式的值，第2个表达式就不会进行求值。

逻辑或表达式：

逻辑与表达式
逻辑或表达式 || 逻辑与表达式

逻辑与表达式：

位或表达式
逻辑与表达式 && 位或表达式

逻辑操作符接受任何标量类型的操作数。两个操作数的类型之间并没有联系，它们各自进行寻常单目转换。逻辑表达式的结果为int类型，其值为0或1，并且不是左值。

AND &&的左操作数首先进行完整的求值。如果左操作数等于0（按照==操作符的含义），则右操作数不再进行求值，直接得出结果0。如果左操作数不等于0，则右操作数进行求值。如果右操作数等于0，则整个表达式的值为0，否则为1。

OR ||的左操作数首先进行完整的求值。如果左操作数不等于0（按照!=操作符的含义），则右操作数不再进行求值，直接得出结果1。如果左操作数等于0，则右操作数进行求值。如果右操作数等于0，则整个表达式的值为0，否则为1。

例子

r = a && b这个赋值相当于

```
if (a == 0) r = 0;
else {
    if (b == 0) r = 0;
    else r = 1;
}
```

r = a || b这个赋值相当于

```
if (a != 0) r = 1;
else {
    if (b != 0) r = 1;
    else r = 0;
}
```

例子

下面是一些逻辑操作符的例子：

a	b	a && b	b是否被求值	a \|\| b	b是否被求值
1	0	0	是	1	否
0	34.5	0	否	1	是
1	"Hello\n"	1	是	1	否
'\0'	0	0	否	0	是
&x	y = 2	1	是	1	否

从语法上说，这两个操作符都是左结合性的，但这并不会对程序员产生太大影响，因为这些操作符在语义上具有完整的结合性，并且其他操作符的优先级都与它们不同。操作符&&的优先级高于||，不过在逻辑表达式中使用括号可以使程序更为清晰、更具可读性。

例子

下面这个表达式

```
a<b || b<c && c<d || d<e
```

与下面这个表达式（它的写法更清晰）相同

```
a<b || (b<c && c<d) || d<e
```

参考：位操作符&和| 第7.6.6节；相等操作符==和!= 第7.6.5节；左值 第7.1节；指针类型 第5.3节；优先级 第7.2.1节；标量类型 第5章；寻常单目转换 第6.3.3节。

7.8 条件表达式

?和:操作符可以组成条件表达式，它的优先级低于双目表达式，并且为右结合性，与双目操作符不同：

条件表达式：

逻辑或表达式

逻辑或表达式 ? 表达式 : 条件表达式

条件表达式具有3个操作数。第1个操作数必须是标量类型，第2个和第3个操作数可以是各种类型，并且将执行寻常单目转换（如果它们都进行求值）。结果的类型取决于第2个操作数和第3个操作数的类型。表7-5显示了传统C所允许的操作数类型，表7-6显示了标准C所允许的操作数类型。条件表达式具有右结合性（相对于第1个和第3个操作数而言），它的结果不是左值，尽管有些标准C之前的编译器把条件表达式的结果当做左值。

表7-5 条件表达式的第2个和第3个操作符（标准C之前）

一种操作数的类型	另一种操作数的类型	结果的类型
算术类型	算术类型	在寻常双目转换之后的类型
结构或联合[a]	同一种结构或联合	该结构或联合类型
指针	同一种类型的指针或0	该指针类型

a 结果的类型具有两个操作数类型的所有限定符。

表7-6 条件表达式的第2个和第3个操作符（标准C）

一种操作数的类型	另一种操作数的类型	结果的类型
算术类型	算术类型	在寻常双目转换之后的类型
结构或联合[a]	兼容的结构或联合	该结构或联合类型
void	void	void
指向限定或未限定版本的类型T_1的指针	指向限定或未限定版本的类型T_2的指针，如果T_1和T_2是兼容的	合成指针类型[a]
指向类型T的指针[b]	限定或未限定版本的void *	void *[a]
任何指针类型	null指针常量	该指针类型[a]

a 结果的类型具有两个操作数类型的所有限定符。

b T必须是个对象或不完整类型。

条件表达式的执行是按照下面的步骤进行的：

1. 第1个操作数被完全求值，并测试是否为零。
2. 如果第1个操作数不等于零，则第2个操作数被求值，它的结果被转换为结果类型，并成

为这个条件表达式的值。第3个操作数并不会被求值。

3. 如果第1个操作数等于零，则第3个操作数被求值，它的结果被转换为结果类型，并成为这个条件表达式的值。第2个操作数不会被求值。

例子

表达式r = a ? b : c相当于

```
if (a != 0) r = b;
else r = c;
```

下面这个表达式

```
a ? b : c ? d : e ? f : g
```

被解释为

```
a ? b : (c ? d : (e ? f : g))
```

例子

在这个例子中，条件表达式的嵌套看上去很有用。signum函数根据它的参数是正值、负值还是零，分别返回1、−1和0：

```
int signum(int x){ return (x > 0) ? 1 : (x < 0) ? -1 : 0; }
```

如果是更加复杂的情况，更好的做法是使用一条或多条if语句。作为一种良好的风格，可以在条件表达式的第1个操作数两边加上括号，但这并非必须。

参考：算术类型 第5章；数组类型 第5.4节；浮点类型 第5.2节；整数类型 第5.1节；左值 第7.1节；指针类型 第5.3节；优先级 第7.2.1节；标量类型 第5章；有符号类型 第5.1.1节；结构类型 第5.6节；联合类型 第5.7节；无符号类型 第5.1.2节；寻常双目转换 第6.3.4节；寻常单目转换 第6.3.3节；void类型 第5.9节。

7.9 赋值表达式

赋值表达式由两个表达式以及中间的一个赋值操作符组成，它们具有右结合性。操作符=称为简单赋值操作符，所有其他赋值符称为复合赋值操作符：

赋值表达式：

条件表达式
单目表达式　赋值操作符　赋值表达式

赋值表达式：下列之一

```
=   +=   -=   *=   /=   %=   <<=   >>=   &=   ^=   |=
```

赋值操作符都具有相同的优先级，并且都具有右结合性（C的其他双目操作符都是左结合性的）。

例子

例如，表达式x *= y = z被当做x *= (y = z)，而不是(x *= y) = z。类似，表达式x = y *= z被当做x = (y *= z)，而不是(x = y) *= z。

赋值操作符的右结合性允许多项赋值表达式具有“明显”的解释。也就是说，表达式a = b = d + 7被解释为a = (b = (d + 7))，因此d + 7的值被赋值给b，然后再赋值给a。

每个赋值操作符要求一个可修改的左值作为它的左操作数，并在这个左值中存储一个新值对它进行修改。这些操作数的区别在于它们是如何计算新值的。赋值表达式的结果不是左值。

参考：可修改的左值 第7.1节；优先级 第7.2.1节。

7.9.1 简单赋值

单个赋值符号=表示简单赋值。它的右操作数的值被转换为左操作数的类型，并存储在左操作数中。表7-7显示了它的操作数允许使用的类型。

表7-7 赋值操作数

左操作数类型	右操作数类型
算术类型	算术类型
结构或联合	兼容的结构或联合
指向T的指针	指向T'的指针，其中T和T'是兼容的
void *	指向T的指针[a]
指向T的指针[a]	void *
任何指针	null指针常量

a 在标准C中，T必须是个对象或不完整类型。

C的最初定义并不允许结构和联合的赋值。有些旧式的编译器仍然具有这个限制。

在标准C中，对于具有类型限定符的操作数，则存在一些其他限制。首先，左操作数不能是const限定的类型。另外：

1. 如果操作数是算术类型，它们可以是限定的，也可以是非限定的。

2. 如果操作数是结构或联合类型，它们必须是兼容类型的限定或非限定版本。例如，这意味着它们的成员必须采用相同的限定。

3. 如果操作数都是对象或函数指针，它们必须是兼容类型的限定或非限定版本。左操作数的类型必须具有右操作数类型的所有限定符。这可以防止const int *指针被赋值为一个int *指针，这种赋值将试图修改常量值。

4. 如果一个操作数是限定版本或非限定版本的void *指针，另一个操作数必须是个指向对象或不完整类型的指针。左操作数的类型必须具有右操作数类型的所有限定符，原因同前。

赋值操作符的结果类型等于左操作数的（未转换且未限定）类型。赋值结果存储在左操作数。它的结果不再是左值。当两个操作数都是算术类型时，就会执行寻常赋值转换，在赋值之前把右操作数的类型转换为左操作数的类型。

简单赋值操作符无法把整个数组的内容复制到另一个数组。数组的名称并不是可修改的左值，因此无法出现在赋值表达式的左边。另外，出现在赋值符号右边的数组名称在赋值之前被转换为指向数组第1个元素的指针（通过寻常转换），因此这种赋值所复制的是个指针，而不是数组的内容。

例子

=操作符可以用于把一个数组的地址复制给一个指针变量：

```
int a[20], *p;
...
p = a;
```

在这个例子中，a是个整型数组，p的类型是“指向整数的指针”。这个赋值导致p指向数组a(的第1个元素)。

可以通过把数组嵌入到一个结构或联合类型中，实现整个数组的复制，这是因为简单赋值可以复制整个结构和联合的内容：

```
struct matrix {double contents[10][10]; };
struct matrix a, b;
…
{
    /* Clear the diagonal elements. */
    for (j = 0; j < 10; j++)
        b.contents[j][j] = 0;
    /* Copy whole 10x10 array from b to a. */
    a = b;
}
```

简单赋值操作符的实现假定右操作数的值和左操作数的值在内存中并不重叠（除非是完全重叠，就像x = x一样）。如果发生了重叠，赋值的行为是未定义的。

参考：算术类型 第5.1.2节；数组类型 第5.4节；寻常赋值转换 第6.3.2节；左值 第7.1节；null指针 第5.3.2节；指针类型 第5.3节；结构类型 第5.6节；类型兼容性 第5.11节；联合类型 第5.7节。

7.9.2 复合赋值

复合赋值操作符可以简单地把“a op= b”理解为“a = a op b”，但表达式a只求值1次。它们所允许的操作数类型取决于具体使用的操作符。表7-8列出了各种可能性。

更准确地说，op=的左右操作数被求值，并且左操作数必须是个可修改的左值。操作符op所指定的操作作用于这两个操作数，包括这个操作符所执行的任何“寻常转换”。它的结果在经过寻常赋值转换之后存储到左操作数所指定的对象。

和简单赋值操作符一样，复合赋值操作符的结果类型等于（未转换的）左操作数的类型。这个结果存储到左操作数中，并且不再是左值。

表7-8 复合赋值表达式的操作数类型

赋值操作符	左操作数	右操作数
*= /=	算术	算术
%=	整数	整数
+= -=	算术	算术
+= -=	指针	整数
<<= >>=	整数	整数
&=	整数	整数
^=	整数	整数
\|=	整数	整数

在早期版本的C中，复合赋值操作符是以相反的顺序书写的，等号出现在操作之前。这可能导致语法的歧义，x=-1可以被解释为x = (-1)或x -= (1)。后来采用的写法消除了这种歧义。由于兼容性的原因，有些非标准的C编译器继续支持这种旧形式，并且将错误地把x=-1解释为x=-(1)，除非等号和减号之间有个空格。

参考：算术类型 第5章；赋值转换 第6.3.2节；浮点类型 第5.2节；整数类型 第5.1节；指针类型 第5.3节；有符号类型 第5.1.1节；无符号类型 第5.1.2节；寻常双目转换 第6.3.4节；寻常单目转换 第6.3.3节。

7.10 线性表达式

逗号表达式由两个用逗号分隔的表达式组成。这里所描述的逗号操作符在语法上具有左结合性，尽管这对于程序员而言并没有什么影响，因为这个操作符本身具有完整的结合性语义。注意，逗号表达式位于C表达式语法树的顶层：

逗号表达式：

赋值表达式
逗号表达式，赋值表达式

表达式：

逗号表达式

逗号操作符的左操作数首先进行完全求值。它不需要产生任何值。即使它产生了一个值，这个值也将被丢弃。然后，它将对右操作数进行求值。逗号表达式的结果的类型和值与右操作数的值经过寻常单目转换之后的类型和值相同。它的结果不再是左值。因此，“r = (a, b, ..., c);”（注意，括号是必要的）这条语句的求值结果是“a; b; ... r = c;”区别在于逗号操作符可以用于表达式语境中，例如用于循环控制表达式。

例子

在for语句中，逗号操作符允许几个赋值表达式组合在一起形成一个表达式，用于单个循环的初始化或者对几个变量进行步进：

```
for( x=0, y=N; x<N && y>0; x++, y--) …
```

逗号操作符具有结合性，所以可以在一个表达式中包含任何数量的由逗号分隔的表达式。子表达式将会按照顺序求值，最后一个子表达式的值将成为整个逗号表达式的值。

例子

过度使用逗号表达式可能会造成混乱，有时候会与逗号的其他用途混淆。例如，下面这个表达式：

```
f(a, b=5,2*b, c)
```

它总是被看成是对接受4个参数的函数f的调用。出现在参数列表中的任何逗号表达式的两边必须加上括号：

```
f(a, (b=5,2*b), c)
```

其他在不加括号的情况下不应该使用逗号表达式的上下文环境还有结构和联合的声明器列表中的字段长度表达式、枚举声明器列表中的枚举值表达式以及声明和初始化值列表中的初始化表达式。逗号还可以作为预处理器宏调用的分隔符使用。

虽然逗号操作符可以保证它的操作数将按照从左到右的顺序进行求值，但逗号字符的其他用法并没有这个保证。例如，在函数调用中，实参表达式的求值顺序并不保证是从左到右进行的。

参考：被丢弃的表达式 第7.13节；枚举类型 第5.5节；for语句 第8.6.3节；函数调用 第7.4.3节；初始化值 第4.6节；左值 第7.1节；宏调用 第3.3节；结构类型 第5.6节；联合类型 第5.7节。

7.11 常量表达式

在有些上下文环境中，C语言允许表达式必须在编译时求值为常量。每个上下文环境都会对它允许使用的表达式的形式施加略有不同的限制。常量表达式可以分为三种类型：

1. 预处理器常量表达式。它作为#if和#elif预处理器控制语句的测试值使用。

2. 整型常量表达式。它用于数组的边界、结构中位段的长度、显式的枚举值以及switch语句中case标签的值。

3. 初始化值常量表达式。它作为静态和外部变量以及（以C99之前）聚合类型的自动变量的初始化值使用。

常量表达式不能包含赋值、增值、减值、函数调用或逗号表达式，除非它们作为sizeof操作符的操作数。其他任何类型的字面值或操作符都可以出现在常量表达式中，但受到下面各小节所讨论的一些其他限制的影响。这些限制是由标准C所添加的。传统C在某些方面可能限制较少。

7.11.1 预处理器常量表达式

预处理器常量表达式必须在编译时求值，它所受到的限制相对较为严格。这种表达式必须为整数类型，并且只能出现整数常量、字符常量以及特殊的defined操作符。在C99中，所有的运算都是使用与目标类型intmax_t或uintmax_t（根据操作数的符号）等价的宿主计算机上的类型完成的。这些类型是在stdint.h中定义的，它们的长度至少为64位。在C99之前，标准C只要求所有的运算使用宿主计算机的long或unsigned long类型完成，当宿主计算机和目标计算机的类型存在明显的区别时，就可能出现问题。

预处理器表达式不能执行任何环境信息查询，除非引用float.h、limits.h、stdint.h等头文件所定义的宏。不允许转换操作，也不能使用sizeof运算符。预处理器中不能出现任何程序变量，即使是用const限定符声明的变量。

例子

下面的代码试图观察目标计算机上的int类型是否大于16位，这种做法是不正确的：

```
#if 1<<16
/* Target integer has more than 16 bits (NOT!)*/
...
#endif
```

事实上，这段代码只测试宿主计算机上的long类型的表示形式（C89）或目标计算机上的intmax_t类型的表示形式（C99）。下面是测试目标类型的长度的正确方法。

```
#include <limits.h>
...
#if UINT_MAX > 65535
    /* target integer has more than 16 bits */
    ...
#endif
```

预处理器必须能够认识字符常量中的转义序列，但它总是允许用户使用源字符集或目标字符集把字符常量转换为整数。这意味着表达式'\n'或'\z'-'a'的值如果出现在预处理器表达式中可能与出现其他地方（例如if语句）不同。对于使用跨计算机编译器的程序员而言，如果宿主计算机和目标计算机的字符集是不同的，就应该注意这个问题。

在宏展开之后，如果预处理器常量表达式包含了任何剩余的标识符，它们每个都被替换为常量0。这可能是个不好的规则，因为如果存在这种标识符，几乎总是意味着出现了程序错误。测试一个名称在预处理器中是否被定义的更好方法是使用defined操作符或#ifdef和#ifndef命令。

编译器可以接受其他形式的预处理器常量表达式，但使用了这种扩展的程序是不可移植的。

参考：类型转换表达式 第7.5.1节；字符常量 第2.7.3节；字符集 第2.1节；defined操作符 第3.5.5节；枚举常量 第5.5节；转义字符 第2.7.5节；float.h 第5.2节；#ifdef和#ifndef 第3.5.3节；

intmax_t 第21.5节； limit.h 第5.1节； sizeof操作符 第7.5.2节； stdint.h 第21章。

7.11.2 整型常量表达式

整型常量表达式用于数组的边界、结构中位段的长度、显式的枚举值以及switch语句中case标签的值。整型常量表达式必须是整数类型，并且可以包含整数常量、字符常量和枚举常量。它可以使用sizeof操作符，并且可以使用任何操作数。它也可以使用类型转换表达式，但只能把算术类型转换为整数类型（除非它们是sizeof的操作数的一部分）。浮点常量只有当它是类型转换的直接操作数或者是sizeof的操作数的一部分时才允许使用。

如果常量表达式不是出现在预处理器命令中，它应该像在目标计算机一样进行求值，包括字符常量的值。

编译器可以接受其他形式的整型常量，包括更为通用的可以转换为整数类型的浮点表达式，但使用这类扩展的程序是不可移植的。有些标准C之前的编译器并不允许在常量表达式中进行任何形式的类型转换。关注这类编译器的移植性问题的程序员应该避免在常量表达式中使用类型转换。

参考：位段 第5.6.5节； 类型转换表达式 第7.5.1节； 枚举类型 第5.5节； 浮点常量 第2.7.2节； sizeof操作符 第7.5.2节； switch语句 第8.7节。

7.11.3 初始化值常量表达式

初始化值中的常量表达式可以包含算术常量表达式和地址常量表达式。

算术常量表达式包括整型常量表达式，并包括更通用的浮点常量（而不仅仅是转换为整数或出现在sizeof表达式中的浮点常量）以及转换为任何算术类型（包含浮点类型）的类型转换表达式。如果一个浮点表达式是在编译时在一个常量表达式中求值的，编译器可能会使用一种比目标环境更精确或范围更广的表示形式。因此，浮点表达式的值在编译时可能与在程序执行时稍有不同。这个规则反映了准确地模拟一种外国浮点实现的困难性。除了这种情况之外，表达式应该像在目标计算机上一样进行求值。

地址常量表达式可以是null指针常量（例如(void *) 0），或者是一个静态或外部对象的地址，或者是一个函数的地址，或者是一个静态或外部对象的地址加上或减去一个整型常量表达式所产生的地址。在形成地址时，可以使用取地址操作符（&）、间接访问操作符（*）、下标操作符（[]）和成员选择操作符（.和->），但不能试图访问任何对象的值。另外，也可以使用转换为指针类型的任何类型转换表达式。

编译器可以接受其他形式的初始化值常量表达式，例如更为复杂的涉及多个地址的地址表达式，但使用这种扩展的程序是不可移植的。

标准C规定了编译器可以在运行时执行初始化，因此可以避免在编译时执行浮点运算。但是，在执行需要访问这些经过初始化的变量的任何代码之前，可能很难执行这种初始化。

例子

下面显示了p和pf的初始化值中所使用的地址常量表达式的例子：

```
static int a[10];
static struct { int f1, f2; } s;
extern int f();
```

```
int i = 3;
…
int *p[] = { &i, a, &a[0],
             (int *)((char *)&a[0]+sizeof(a)),
             &s.f2 };
int (*pf)() = &f;
```

参考：地址操作符& 第7.5.6节；数组类型 第5.4节；初始化值 第4.6节；sizeof操作符 第7.5.2节；结构类型 第5.6节。

7.12 求值的顺序

一般而言，编译器可以重新排列表达式的求值顺序。这种重新排列可能出现在对函数调用的参数进行求值，或者对一个双目操作符的两个操作数进行求值，采用某种与最直观的从左到右方式不同的求值顺序。双目操作符+、*、&、^和|被认为具有完整的结合性，它们的两个操作数符合交换律，编译器可以充分利用这个特征。例如，编译器可以按照(a+d)+(b+c)的顺序对表达式(a+b)+(c+d)进行求值（假定所有的变量都为相同的算术类型）。

对于&、^和|操作符而言，如果它们作用于无符号操作数，也满足交换律和结合性。但是，如果它们的操作数是有符号数，则这个结论未必正确，具体取决于有符号整数的表示形式。对于*和+，这个结论并不能成立，因为它们在有些求值顺序下可能会发生溢出，但在其他求值顺序下则不会发生溢出。但是，编译器总是可以利用这个特点。对涉及这些操作符的表达式进行重新排列时，必须保证不改变操作数的隐式类型转换。

例子

为了控制求值的顺序，程序员可以使用赋值到临时变量的方法。但是，一个经过良好优化的编译器甚至会对下面这样的计算进行重新排列：

```
int temp1, temp2;
…
/* Compute q=(a+b)+(c+d), exactly that way. */
temp1 = a+b;
temp2 = c+d;
q = temp1 + temp2;
```

例子

在下面这个例子中，两个表达式并不等价，编译器无法用其中一个代替另一个，虽然这两个表达式的差异只是交换了某些加法的顺序而已：

```
(1.0 + -3) + (unsigned) 1; /* Result is -1.0 */
1.0 + (-3 + (unsigned) 1); /* Result is large */
```

第1个赋值非常简单直观，它产生了预期的结果。第2个赋值产生一个很大的结果，因为寻常双目转换导致有符号数−3被转换为一个很大的无符号值$2^n - 3$，其中n是表示无符号值的位数。这个结果随后与无符号值1相加，其结果转换为浮点表示形式，并与1.0相加，产生的结果是用某种浮点形式表示的值$2^n - 1$。这个结果可能是程序员所预期的，也有可能不是，但编译器不能因为重新排列加法运算而造成这种混乱。

根据语言的定义，编译器也可以重新排列浮点表达式。但是，取决于操作数的具体值，浮

点表达式的求值顺序可能会对结果的精确性产生很大影响。由于编译器无法预测操作数的值，数值分析家倾向于让编译器总是按照浮点表达式的书写顺序进行求值。这样，程序员就可以控制求值的顺序。

对函数调用中的实际参数进行求值时，实参和函数表达式的求值顺序是未指定的，但是其效果相当于选择一个参数，对它完整求值，然后选择另一个参数，再对它完整求值，直到所有的参数都求值完毕。双目表达式操作符的每个操作数以及表达式a[i]中的a和i也存在这样的选择和限制。

例子

在这个例子中，变量x是个字符指针数组，它被看成是个字符串数组。变量p是个指向字符指针的指针，它被看成是个指向字符串的指针。这条if语句的目的是判断p所指向的字符串（称为s1）与它后面的那个字符串（称为s2）是否相等（在此之后，p将指向这两个字符串之后的那个字符串）。

```
char *x[10], **p=x;
…
if ( strcmp(*p++, *p++) == 0 ) printf("Same.");
```

当然，让一个表达式中的同一个变量的副作用出现两次并不是一种好的编程风格，因为这些副作用的顺序并没有被定义。但是，有些自作聪明的程序员则声称副作用不会产生影响，因为提供给strcmp函数的这两个字符串的顺序并没有关系。

序列点

在标准C中，如果一个对象在连续的序列点之间被修改的次数超过1次，其结果是未定义的。所谓序列点，就是在程序的执行序列中的某个点，此时程序执行的以前所有副作用都已经发生，但下一次的副作用还没有发生。序列点出现在下列情况：

- 在一个完整表达式的尾部，即初始化值、表达式语句、return语句中的表达式以及条件、迭代或switch语句的控制表达式（包括for语句中的每个表达式）
- &&、||、?:或逗号操作符的第1个操作数之后
- 在函数调用中实参和函数表达式的求值之后

根据这个规则，表达式++i*++i的值是未定义的，就像前一个strcmp例子一样。

参考：加法操作符+ 第7.6.2节；双目操作符 第7.6节；位AND操作符& 第7.6.6节；位OR操作符| 第7.6.8节；位XOR操作符^ 第7.6.7节；逗号操作符 第7.10节；条件表达式?: 第7.8节；条件语句 第8.2节；函数调用 第7.4.3节；初始化值 第4.6节；迭代语句 第8.6节；逻辑和&&和逻辑或|| 第7.7节；乘法操作符* 第7.6.1节；return语句 第8.9节；strcmp函数 第13.2节；寻常双目转换 第6.3.4节。

7.13 被丢弃的值

在三种上下文环境中，可以出现表达式，但它的值并不会被使用：

1. 表达式语句
2. 逗号表达式的第1个操作数

3. for语句的初始化表达式和增值表达式

在这些上下文环境中，我们称这些表达式的值被丢弃。

当一个没有副作用的表达式的值被丢弃时，编译器可能会认为出现了编程错误，并发出一个警告。可以产生副作用的操作包括赋值和函数调用。如果一条被丢弃的表达式的主操作符并没有副作用，编译器也可能会发出警告。

例子

```
extern void f();
f(x);      /* These expressions do not */
i++;       /* justify any warning about */
a = b;     /* discarded values.  */
```

这些语句尽管都是合法的，但可能会导致编译器发出警告：

```
extern int g();
g(x);             /* The result of g is discarded. */
x + 7;            /* Addition has no defined side effects. */
x + (a *= 2);     /* The result of the last operation to be
                     performed, "+", is discarded. */
```

程序员可以通过转换为void类型，表示有意丢弃表达式的值，避免编译器对此发出警告：

```
extern int g();
(void) g(x);    /* Returned value is purposely discarded */
(void)(x + 7); /* This is pretty silly, but presumably
                  the programmer has a purpose. */
```

当一个函数调用的值被丢弃时，编译器一般并不会发出警告，因为在传统上不返回结果的函数必须被声明为“返回int类型的函数”。尽管标准C向编译器提供了更多信息，但编译器厂商还是选择与旧式的代码保持兼容。

如果编译器判断被丢弃的表达式的主操作符没有副作用，它可能会选择不产生该操作符的代码（此时它的操作数成为被丢弃的值，并且可能递归地受这种处理的控制）。

参考：赋值 第7.9节；类型转换 第7.5.1节；逗号操作符 第7.10节；for语句 第8.6.3节；函数调用 第7.4.3节；表达式语句 第8.2节；void类型 第5.9节。

7.14 内存访问的优化

作为一个基本规则，编译器可以生成与程序所书写的计算行为相同的任何代码。编译器被明确授予了一些权利对代码进行重新排列，如第7.12节所述。当表达式不具有副作用并且它的值被丢弃时，编译器还可以选择不生成该表达式的代码，如第7.13节所述。

例子

有些编译器可能会发现这样的代码，例如它并不需要多次访问内存，或者并不总是根据程序所指定的顺序访问内存。例如，如果某个数组元素被多次引用，则编译器可以进行合理的安排，只读取它一次以提高速度。它可能对下面的代码进行改写：

```
int x,a[10];
…
x = a[j] * a[j] * a[j];   /* Cube the table entry. */
```

使它就像下面这样的写法一样被执行：

```
int x,a[10];
register int temp;
…
temp = a[j];
x = temp * temp * temp;  /* Cube the table entry. */
```

对于大多数应用程序而言，包括几乎所有的可移植应用程序，像这样的优化技巧是非常实用的，因为程序的速度可以通过一两个因素得到改善，并且不会影响程序的有效计算行为。但是，用C语言编写中断处理函数以及一些依赖于计算机的程序时，这可能会产生问题。在这种情况下，程序员应该使用标准C的类型限定符volatile来控制一些内存访问。

参考：volatile 第4.4.5节。

7.15 C++兼容性

sizeof表达式的改变

在C++中，在表达式中声明类型是非法的，例如在类型转换表达式或sizeof表达式中。另外，有些sizeof表达式的值在C和C++中可能不同，这是由于作用域的变化以及字符常量的类型不同。

例子

```
i = sizeof(struct S { … }); /* OK in C, not in C++ */
```

例子

sizeof(T)的值在T被重定义的有些情况下可能会不同。

sizeof('a')的值在C中是sizeof(int)，但在C++中是sizeof(char)。

对于枚举常量e，sizeof(e)的值在C中是sizeof(int)，但在C++中可能不同。

参考：字符常量 第2.8.5节；枚举类型 第5.13.1节；作用域的差别 第4.9.2节；sizeof 第7.5.2节。

7.16 练 习

1. 下面哪些表达式在传统C中是合法的？对于那些合法的表达式，表达式的类型是什么？假设f的类型是float，i的类型是int，cp的类型是char *，ip的类型是int *。

(a) `cp+0x23`	(f) `f==0`
(b) `i+f`	(g) `!ip`
(c) `++f`	(h) `cp && cp`
(d) `ip[i]`	(i) `f%2`
(e) `cp?i:f`	(j) `f+=i`

2. 假设p1和p2的类型是char *。重新编写下面这两条语句，不使用增值或减值操作符：

```
(a) *++p1=*++p2;
(b) *p1--=*p2--;
```

3. “位掩码”是一种由指定的二进制0和1序列所组成的整数。编写一些宏，产生下面这些位掩码。如果宏的参数是常量，结果也应该是常量。可以假设整数采用补码表示，但这些宏不应该依赖于整数的位数或者计算机所采用的是大端字节顺序还是小端字节顺序。

(a) low_zeroes(n)，一个字的低端n个位都是0，其他的位都是1。

(b) low_ones(n)，一个字的低端n个位都是1，其他的位都是0。

(c) mid_zeroes(width, offset)，一个字的低端offset个位为1，接下来width个位为0，其余的位都为1。

(d) mid_ones(width, offset)，一个字的低端offset个位为0，接下来width个位为1，其余的位都为0。

4. j++ == ++j是否为合法的表达式？j++&&++j呢？如果j一开始的值是0，每个表达式的结果是什么？

5. 下面这个表列出了简单赋值表达式左右两边的类型组合。哪些组合在标准C中是允许的？

	左侧类型	右侧类型
(a)	short	signed short
(b)	char *	const char *
(c)	int (*)[5]	int (*)[]
(d)	short	const short
(e)	int (*)()	signed (*)(int x, float d)
(f)	int *	t * （其中：typedef int t）

6. 如果变量x的类型是struct{int f;}，变量y的类型是另一个独立的定义struct{int f;}，在标准C中，x=y是否合法？



第8章
语　句

C语言提供了在大多数代数型编程语言中可以找到的通常类型的语句，包括条件语句、循环语句和独特的goto语句。在讨论基本的语法之后，我们将描述每一种语句：

语句：

- 表达式语句
- 标签语句
- 复合语句
- 条件语句
- 迭代语句
- switch语句
- break语句
- continue语句
- return语句
- goto语句
- null语句

条件语句：

- if 语句
- if-else 语句

迭代语句

- do语句
- while语句
- for语句

8.1 语句的基本语法规则

尽管C的语句对于那些熟悉类似ALGOL语言的程序员而言并不会陌生，但它还是存在一些区别，常常会引起混淆并导致错误。

与Pascal和Ada一样，C的分号一般出现在连续的语句之间。但是，C语言的分号并不是语句分隔符。在有些语句中，分号就是语句的一部分。唯一不需要分号的语句是复合语句（或称代码块），它是位于一对花括号（{}）内部，而不是位于其他编程语言所使用的begin和end关键字之间：

```
a = b;
{ b = c; d = e; }
x = y;
```

C语句的另一个规则是条件或迭代语句的“控制”表达式必须出现在括号中。控制表达式的后面并没有特殊的关键字，例如“then”、“loop”或“do”。语句的剩余部分紧随控制表达式

之后：

```
if (a<b) x=y;
while (n<10) n++;
```

最后，其他语言的赋值语句就是C的赋值表达式。它可以作为更复杂的表达式的一部分，也可以在后面跟个分号，表示它本身就是一条单独的语句：

```
if ((x=y)>3) a=b;
```

参考：赋值表达式 第7.9节；复合语句 第8.4节；条件语句 第8.5节；迭代语句 第8.6节。

8.2 表达式语句

任何表达式都可以在后面跟个分号，使它成为一条语句：

表达式语句：

表达式 ;

这种语句的执行方式是对表达式进行求值，然后丢弃它所产生（如果有）的结果。

只有当表达式的求值具有副作用时，表达式语句才是有用的，例如把一个值赋值给一个变量或者执行输入或输出。这种表达式通常是赋值表达式、增值或减值表达式或者函数调用。

例子

```
speed = distance / time;  /* assign a quotient */
++event_count;            /* Add 1 to event_count.*/
printf("Again?");         /* Call the function printf.*/
pattern &= mask;          /* Remove bits from pattern */
(x<y) ? ++x : ++y;        /* Increment smaller of x and y */
```

尽管是最后一条语句合法的，但它可以用if语句写成下面这种更清晰的形式：

```
if (x < y) ++x;
else ++y;
```

对于没有副作用并且求值结果将被丢弃的表达式，编译器可以不对它进行求值，或者不对它的一部分进行求值（参见第7.13节）。

参考：赋值表达式 第7.9节；丢弃表达式 第7.13节；表达式 第7章；函数调用第7.4.3节；增值表达式 第7.5.8节、第7.4.4节。

8.3 标签语句

标签可以用于标记任何语句，使程序的控制可以通过goto或switch语句转移到这类语句。标签共分为3种。命名标签可以出现在任何语句中，一般和goto语句一起使用。case标签或default标签只能出现在switch语句体内部的语句中：

标签语句：

标签：语句

标签：

命名标签
case标签
default标签

标签不能单独出现，必须和语句一起出现。如果需要把标签放在单独的位置（例如在复合语句的最后），可以把它放在空语句中。在C99中，语句和声明的位置可以是任意的，标签无法直接应用于声明，它必须与声明之前的一条空语句放在一起。

命名标签将在描述goto语句时讨论。case和default标签将在介绍switch语句时讨论。

参考：goto语句 第8.10节；空语句 第8.11节；switch语句 第8.7节。

8.4 复合语句

复合语句是由一对花括号所包围的零个或多个声明和语句列表所组成。在C99中，声明和语句的出现位置可以是任意的。在以前版本的C中，声明必须出现在语句之前。

复合语句：

{ 声明或语句列表$_{opt}$ }

声明或语句列表：

声明或语句
声明或语句列表 声明或语句

声明或语句：

声明
语句

复合语句可以出现在语句可以出现的任何地方。它引入了一个新的作用域（或代码块），这个作用域适用于复合语句内部的任何声明或复合字面值。正常情况下，复合语句是按照顺序逐条执行它的每个声明和语句的。当最后一个声明或语句执行完成后，复合语句的执行便结束。在复合语句结束之前用goto、return、contitue或break语句跳出复合语句是可行的。除了从开始位置进入复合语句之外，还可以通过goto或switch语句跳转到该复合语句内部的一个标签。跳入或跳出复合语句可能会影响复合语句内部的声明，这个问题将在下一节讨论。

参考：auto存储类别 第4.3节；break和continue语句 第8.8节；声明 第4章；goto语句 第8.10节；register存储类别 第4.3节；return语句 第8.9节；作用域 第4.2.1节。

复合语句内部的声明

在复合语句或其他代码块内部声明的标识符称为代码块层次的标识符，这种声明称为代码块层次的声明。代码块层次的标识符的作用域是从它的声明位置直到代码块的末尾。这个标识符在这个作用域中是可见的，除非被内层代码块的同一个标识符声明所隐藏。在代码块中声明标识符通常是一种良好的编程实践，因为限制变量的作用域可以使程序更容易理解。

对于在代码块中声明并且未指定存储类别的标识符，如果它是函数类型，它的存储类别默认为extern。其他类型的标识符则默认为auto。当函数是在代码块中声明时，它的存储类别只能是extern，其他存储类别都是非法的。

如果一个变量或函数的存储类别在代码块中被声明为extern，就不会为它分配内存，并且不允许使用初始化表达式。这个声明引用了一个外部变量或者在其他地方所定义的函数，它们可以位于同一个源文件，也可以位于不同的源文件。

除了可变长度的数组之外，如果一个在代码块中声明的变量的存储类别被指定为auto或

register，则每次进入这个代码块时，这个变量都会分配未定义的值，并在每次离开代码块后被销毁。也就是说，这个变量的生命期是整个代码块，而不仅仅是从它的声明点开始。如果这个变量的声明具有一个初始化值表达式，则控制流每次进入这个代码块时，都会对这个初始化值表达式进行求值，因此这个变量每次都会进行初始化。如果一条goto语句把控制从复合语句的内部退回到这个声明之前的位置，这个变量的初始化在正常情况下只发生一次，但在C99中则允许发生多次。如果一条goto或switch语句把控制跳入到复合语句中该变量声明之后的位置，它的初始化值可能并没有被求值，这个变量的值可能仍然处于未定义状态。自动代码块层次的标识符并不会把某次执行时的值保留到下一次执行的时候。

在C99中，在代码块中声明的可变长度的数组并不像其他自动变量一样是在代码块的入口分配内存的。当程序的控制遇到它的声明并且对它的长度表达式进行求值时，才会为它分配内存。当程序的控制离开这个代码块时，就会销毁分配给它的内存。因此，它的生命期和作用域是相同的。可变长度的数组无法进行初始化。从作用域的外部跳入到数组作用域（即在声明之后）是非法的。从作用域内部跳回到声明之前的一个位置是允许的。在这种情况下，数组就会被销毁并重新分配内存，并可能具有新的长度。代码块中所有可变长度的数组都遵循先销毁后分配的原则，因此可以在过程调用堆栈上为它们分配内存。

如果一个在代码块中声明的变量指定了存储类别static，它将只分配一次，在程序执行之前，就像其他任何静态变量一样。如果在声明时指定了一个初始化表达式，则这个初始化值（必须是个常量）只求值一次，就在程序执行之前，并且这个变量在复合语句的不同次执行之间可以保持它的值。在C99中，它的初始化值也必须是常量。

例子

在下面的代码片段中，如果语句标签L：是从复合语句外部跳入的目标，这些代码很可能出错，因为变量sum将不会被初始化。而且，它并没有办法区分这种跳入是不是在根本没有检查外层函数的整个函数体的情况下发生。

```
{
        extern int a[100];
        int i, sum = 0;
        …
    L:
        for (i = 0; i < 100; i++)
            sum += a[i];
        …
}
```

例子

当非标签复合语句作为switch语句的语句体时，它在正常情况下是无法被执行的。它只能通过把控制转移到它内部的某条标签语句来执行。因此，在这种复合语句的开始处对auto和register变量的初始化永远被不会执行，肯定是错误的。

```
switch (i) {
          int sum = 0; /* ERROR! sum is NOT set to 0 */
case 1: return sum;
default: return sum+1;
}
```

参考：auto存储类别 第4.3节；extern存储类别 第4.3节；goto语句 第8.10节；初始值 第4.2.8节；初始化值 第4.6节；register存储类别 第4.3节；作用域 第4.2.1节；static存储类别 第4.3节；switch语句 第8.7节；可变长度的数组 第5.4.5节；可见性 第4.2.2节。

8.5 条件语句

条件语句具有两种形式，取决于它是否使用else分句。C并不使用then关键字作为if语句的一部分：

条件语句：

```
if语句
if - else语句
```

if语句：

```
if (表达式) 语句
```

if－else语句：

```
if (表达式) 语句  else 语句
```

对于每种形式的if语句，括号内的表达式首先进行求值。如果这个值是非零值（第8.1节），则括号后面的语句立即被执行。如果控制表达式是零值，并且存在else分句，则关键字else后面的那条语句将被执行。但是，如果控制表达式为零并且不存在else分句，程序的控制就会立即转称到该条件语句之后的那条语句。

在C99中，整条if语句形成了它自己的代码块作用域。它的子语句也是如此，即使它们并不是复合语句。这是为了限制作为使用复合字面值或类型名称的副作用而创建的对象和类型的作用域。

参考：复合字面值 第7.4.5节；控制表达式 第8.1节；类型名称 第5.12节。

8.5.1 多路条件语句

多路决策可以用一系列的if－else语句来表示，其中每条if语句（除了最后一条之外）的else分句中有另一条if语句。例如下面的系列：

```
if (expression1)
    statement1
else if (expression2)
    statement2
else if (expression3)
    statement3
…
else
    statementn
```

例子

下面是一个3路决策：如果signum的参数小于0就返回－1；如果大于0就返回1；在其他情况下返回0：

```
int signum(int x)
{
```

```
    if (x > 0) return 1;
    else if (x < 0) return -1;
    else return 0;
}
```

读者可以把这个版本的signum与第7.8节使用特定类型的多路决策版本的signum函数进行比较。switch语句处理特定类型的多路决策，其中表达式的值与一组固定的常量进行比较。

参考：switch语句 第8.7节。

8.5.2 悬疑的else问题

由于条件语句可以包含其他条件语句，所以可能会产生歧义。在有些情况下，可能无法分辨一条else语句应该属于哪一条if语句。这种歧义性是通过一种约定俗成的方式解决的：在可能的情况下，else语句总是属于它前面最内层的那条if语句。

例子

为了显示这种歧义性，下面这个例子使用了具有误导风格的缩进：

```
if ((k >= 0) && (k < TABLE_SIZE))
    if (table[k] >= 0)
        printf("Entry %d is %d\n", k, table[k]);
else printf("Error: index %d out of range.\n",k);
```

粗心的读者可能会以为这条else语句属于外层的if语句。也就是说，当下面这个测试的值为假时，应该打印出错误信息。

```
(k >= 0) && (k < TABLE_SIZE)
```

但是，如果我们把最后一条错误信息修改为：

```
else printf("Error: entry %d is negative.\n", k);
```

此时，程序员的意图似乎是当table[k] >= 0这个测试为false时执行else语句。在前面代码片段中，第2种解释是正确的，而第1种解释是错误的。为了使前面的代码段符合第1种解释，可以像下面这样使用复合语句：

```
if (k >= 0 && k < TABLE_SIZE) {
    if (table[k] >= 0)
        printf("Entry %d is %d\n", k, table[k]);
}
else printf("Error: index %d out of range.\n", k);
```

为了避免混淆，第2种解释也可以使用复合语句：

```
if (k >= 0 && k < TABLE_SIZE) {
    if (table[k] >= 0)
        printf("Entry %d is %d\n", k, table[k]);
    else printf("Error: entry %d is negative.\n",k);
}
```

如果总是对if语句所控制的语句使用花括号，就可以消除这种混淆。但是，这种保守的规则可能导致程序中充斥着不必要的花括号。为了在混淆和杂乱之间取得平衡，我们可以采取的一种妥协风格是当if所控制的语句是表达式语句或空语句之外的语句时，都应该加上花括号。

参考：复合语句 第8.4节；表达式语句 第8.2节；空语句 第8.11节。

8.6 迭代式语句

C提供了3种迭代式语句：

迭代式语句：

```
while语句
do语句
for语句
```

while语句在每次执行之前测试退出条件。do语句在每次执行之后测试退出条件。for语句提供了一种特殊的语法，可以方便地初始化并更新1个或多个控制变量，以及测试退出条件。嵌入在迭代语句中的语句又称为迭代语句的语句体。

在C99中，每条迭代语句形成了它自己的代码块作用域，它的子语句也是如此，即使它们并不是复合语句。这是为了限制作为使用复合字面值或类型名称的副作用而创建的对象和类型的作用域。

参考：复合字面值 第7.4.5节；控制表达式 第8.1节；类型名称 第5.12节。

8.6.1 while语句

C并不使用do关键字作为while语句的语法的一部分：

while语句：

```
while (表达式)  语句
```

在执行while语句时，首先对控制表达式进行求值。如果结果是真（非零），则语句被执行。整个过程将会反复进行，每次都对控制表达式进行求值，如果值为真，就执行语句。表达式每次的值可能会发生变化，这是由于语句或表达式的副作用所致。

当控制表达式的求值结果为假（零）或者由于return、goto或break语句使程序的控制离开while语句体时，while语句的执行便结束。continue语句也可以更改while语句内部的执行流。

例子

下面这个函数使用一个while循环计算整数base的非负整数exponent次方（不检查溢出）。它所使用的方法是反复乘以base，并用二进制记法对exponent进行解码，以确定在什么时候把base乘到结果中。

为了理解这种方法为什么可行，注意while循环维护了一个不变性条件，就是正确答案是result乘以base的exponent次方的结果。当exponent最终为0时，这个条件就分解完毕，表示result包含了正确的结果。

```
int pow(int base, int exponent)
{
    int result = 1;
    while (exponent > 0) {
        if ( exponent % 2 ) result *= base;
        base *= base;
        exponent /= 2;
    }
    return result;
}
```

例子

while循环常常可以使用一条空语句作为它的语句体：

```
while ( *char_pointer++ );
```

在上面的代码中，++操作符使一个字符指针不断向前移动，直到遇见一个null字符。这时，这个指针指向null字符之后的位置。这是查找字符串尾部的一种紧凑方式（注意，测试表达式被解释为*(char_pointer++)，而不是(*char_pointer)++，后者用于增加char_pointer所指向的字符的值）。

例子

另一种常见的用法是使用两个指针复制一个字符串：

```
while ( *dest_pointer++ = *source_pointer++ );
```

它将不断复制字符，直到遇见表示字符串终止标志的null字符（这个字符也会被复制）。当然，在编写这样的代码时，程序员应该确保目标区域有足够的空间容纳所有被复制的字符。

参考：break和continue语句 第8.8节；控制表达式 第8.1节；goto语句 第8.10节；空语句 第8.11节；return语句 第8.9节。

8.6.2 do语句

do语句与while语句的不同之处在于do语句至少执行循环体1次，而while语句可能连1次也不执行：

do语句：

```
do 语句 while (表达式) ;
```

在执行do语句时，首先执行的是它所包含的语句，然后对控制表达式进行求值。如果这个值为真（非零），整个过程将会重复，再次执行语句，然后再测试控制表达式。如果测试结果仍然为真，就再次重复这个过程。

当控制表达式的求值结果为假（零）或者由于return、goto或break语句使程序的控制离开do语句体时，do语句的执行便结束。continue语句也可能更改do语句内部的执行流。

C的do语句在功能上与Pascal的“repeat-until”语句相似。C的do语句的不同寻常之处在于它在控制表达式的值为假时终止执行，而Pascal的repeat-until语句在控制表达式为真时终止执行。C在这方面更为一致：C的所有迭代结构（while、do和for）都是在控制表达式为假时终止执行。

例子

下面这个程序片段读取并处理字符。当它处理了一个换行符时就终止执行：

```
int ch;
do
    process( ch = getchar());
while (ch != '\n');
```

把计算过程移动到while语句的控制表达式中可以取得相同的效果，但代码看上去没那么清晰：

```
int ch;
while( ch = getchar(ch),
       process(ch),
       ch != '\n' ) /*empty*/ ;
```

例子

语句体为空语句的do语句也是可以的：

```
do ; while (expression);
```

但是，更常见的做法是使用while语句编写这种循环：

```
while (expression);
```

参考：break和continue语句 第8.8节；控制表达式 第8.1节；goto语句 第8.10节；null语句 第8.11节；return语句 第8.9节；while语句 第8.6.1节。

8.6.3 for语句

C的for语句要比其他大多数语言中的“增值并测试”语句更为通用。在解释了for语句的执行之后，我们将提供几个例子，说明它的用法：

for语句：

```
for  for表达式  语句
```

for表达式：

```
(初始化分句opt ; 表达式opt ; 表达式opt)
```

初始化分句：

```
表达式
声明                                    (C99)
```

for语句由关键字for、位于一对括号内部由分号分隔的3个表达式以及一条语句所组成。括号内的每个表达式都是可选的，可以被省略，但是分隔这些表达式的分号以及它们外边的括号必须保留。

一般情况下，第1个表达式用于初始化循环变量，第2个表达式测试循环应该继续还是终止，第3个表达式用于更新循环变量（例如增加它的值）。但是，这些表达式在原则上可以执行在for结构的框架内有用的任何计算。for语句是按照下面的方式执行的：

1. 如果初始化分句是个表达式，则对它进行求值，然后丢弃求值结果。如果初始化分句是个声明（C99），则被声明的变量就进行初始化。如果不存在初始化分句，就跳过这一步骤。

2. 如果第2个表达式存在，它就像控制表达式一样进行求值。如果结果为0，for语句的执行便结束。否则（如果值不为0或者第2个表达式被省略），就进入步骤3。

3. for的语句体被执行。

4. 如果存在第3个表达式，这个表达式就进行求值，其值被丢弃。

5. 返回到步骤2。

当第2个表达式（控制表达式）的值为0或者程序的控制被return、goto或break语句跳转到for语句的外部时，for语句的执行便告终止。在for语句中执行continue语句将会把控制转移到步骤4。

在C99中，for语句形成了它自己的作用域，它的子语句也是如此，即使它们并不是复合语句。这是为了限制作为使用复合字面值或类型名称的副作用而创建的对象和类型的作用域。另外，for循环的第1个表达式也可以被一个声明所代替，后者可以声明并初始化1个或多个循环控制变量。这些变量的作用域一直延续到for语句的结束，包括循环控制的第2个和第3个表达式。在编写for循环时需要这种变量是很常见的，限制它们的作用域可以使C编译器进行更好的优化。

参考：break和continue语句 第8.8节；复合字面值 第7.4.5节；控制表达式 第8.1节；丢弃表达式 第7.13节；goto语句 第8.10节；return语句 第8.9节；类型名称 第5.12节；while语句 第8.6.1节。

8.6.4 使用for语句

例子

一般而言，for语句的第1个表达式用于初始化一个变量，第2个表达式按照某种方式测试这个变量，第3个表达式向着某个目标修改这个变量。例如，为了打印出整数0-9以及它们的平方，可以编写下面的代码：

```
int j;
...
for (j = 0; j < 10; j++)
   printf("%d %d\n", j, j*j);
```

这里，第1个表达式对j进行初始化，第2个表达式测试它是否到达了10（如果是，这个循环就终止），第3个表达式把j的值增加1。

在C99中，变量j可以在循环中声明，因此它的作用域限制在这个循环中：

```
for (int j = 0; j < 10; j++)
   printf("%d %d\n", j, j*j);
```

例子

下面是在C中编写“永不终止”的循环的两种方式（又称无限循环）：

```
for ( ; ; ) 语句
while (1) 语句
```

这种循环可以通过循环体内的break、goto或return语句终止。

例子

前面说明while语句用法的pow函数可以改写为使用for语句：

```
int pow(int base, int exponent)
{
   int result = 1;
   for (; exponent > 0; exponent /= 2) {
      if ( exponent % 2 )
         result *= base;
      base *= base;
   }
   return result;
}
```

这种形式强调这个循环是由exponent变量反复除以2而趋向0的过程所控制的。注意循环变量exponent仍然必须在for语句的外部声明。这条for语句并没有包含任何变量的声明。一种常见的编程错误的是忘了声明在for语句中所使用的变量（例如i和j），结果导致其他地方所声明的名称为i或j的变量意外地被这个循环所修改。

例子

下面是一个简单的排序函数，它使用了插入排序算法。

```
void insertsort(int v[], int n)
{
```

```
    register int i, j, temp;
    for (i = 1; i < n; i++) {
        temp = v[i];
        for (j = i-1; j >= 0 && v[j] > temp; j--)
            v[j+1] = v[j];
        v[j+1] = temp;
    }
}
```

外层的for循环从1到n（包括1，不包括n）对i进行计数。在每一步中，元素v[0]到v[i-1]已经排好序，元素v[i]到v[n-1]仍然有待排序。内层的循环从i-1开始对j进行倒计数，一次移动一个数组元素，直到找到v[i]需要插入的正确位置（这是这种算法称为插入排序的原因）。对于非常庞大的未排序数组，这种算法的效率并不高，因为在最坏的情况下，这种排序的时间复杂度是n*n（即$O(n^2)$）。

例子

只要简单地在前两个循环的外面包装第3个循环，并在insertsort使用常量1的几个地方引入gap，可以把插入排序的效率从$O(n^2)$提高到$O(n^{1.25})$。下面这个排序函数使用了希尔排序算法，它与Kernighan和Richie的《The C Programming Language》中作为例子出现的shell排序函数相似。但是，我们在这里根据3种方法对它进行了修改，其中两个是由Knuth和Sedgewich（参见前言）所建议的，目的是提高这种算法的速度：

```
void shellsort(register int v[], int n)
{
    register int gap, i, j, temp;
    gap = 1;
    do (gap = 3*gap + 1); while (gap <= n);
    for (gap /= 3; gap > 0; gap /= 3)
        for (i = gap; i < n; i++) {
            temp = v[i];
            for (j=i-gap; (j>=0)&&(v[j]>temp); j-=gap)
                v[j+gap] = v[j];
            v[j+gap] = temp;
        }
}
```

改进之处在于：（1）在原来的shell函数中，gap的值从n/2开始，每次在外层循环中除以2。在这个版本中，gap被初始化为（1，4，13，40，121，…）这个序列中不大于n的最大值，并且每次经过外层循环时，gap的值被除以3。这个改进使这个算法的速度提高了20%～30%（gap初始值的这种选择方法证明比使用n作为初始值更为优越）。（2）内层循环中的赋值从3个减少到1个。（3）增加了register和void存储类别。在有些编译器中，register声明可以极大地提高性能（有提高40%的例子）。

例子

for语句并不一定只对整数值进行计数。下面这个循环扫描一个结构链表，它的循环变量是个指针：

```
struct intlist {
    struct intlist *link;
    int data;
};

void print_duplicates(struct intlist *p)
```

```
{
    for (; p; p = p->link) {
        struct intlist *q;
        for (q = p->link; q; q = q->link)
            if (q->data == p->data) {
                printf("Duplicate data %d", p->data);
                break;
            }
    }
}
```

intlist结构用于实现一个记录链表，每个记录包含了一些数据。有了一个这样的链表之后，print_duplicates函数打印出链表中每个冗余记录的数据。第1条for语句使用形式参数p作为它的循环变量。这条for语句扫描这个特定的链表。当它遇到一个null指针时，这个循环便终止。对于每个记录，它后面的所有记录都是由内层的for语句所检查的，后者按照同样的方式使用指针q扫描这个链表。

参考：指针类型 第5.3节；register存储类别 第4.3节；选择操作符-> 第7.4.2节；结构类型 第5.6节；void类型 第5.9节。

8.6.5 多个控制变量

有时候，用多个变量控制一个for循环较为方便。在这种情况下，逗号操作符具有特殊的用处，因为它可以把几个赋值表达式合并到一个表达式中。

例子

下面这个函数通过修改link链反转一个链表：

```
struct intlist { struct intlist *link; int data; };

struct intlist *reverse(struct intlist *p)
{
    struct intlist *here, *previous, *next;
    for (here = p, previous = NULL ;
         here != NULL ;
         next = here->link, here->link = previous,
           previous = here, here = next) /*empty*/ ;
    return previous;
}
```

例子

下面这个函数string_equal接受两个字符串为参数。如果它们相等，这个函数返回1，否则返回0。

```
int string_equal(const char *s1, const char *s2)
{
    char *p1, *p2;
    for (p1=s1, p2=s2; *p1 && *p2; p1++, p2++)
        if (*p1 != *p2) return 0;
    return *p1 == *p2;
}
```

这条for语句通过两个指针变量同时扫描这两个字符串。表达式p1++，p2++使每个指针指向下一个字符。如果发现两个字符串不同，return语句就终止整个函数的执行，并返回0。如果在

任一字符串中遇到了null字符（由表达式*p1 && *p2所确定），则循环正常终止，第2条return语句判断两个字符串是否在相同的位置以null字符结束（如果使用表达式*p1代替*p1 && *p2，这个函数仍然可以正确地完成任务。而且，它的速度还能更快一些，虽然它看上去有点不对称）。

参考：break和continue语句 第8.8节；逗号操作符 第7.10节；指针类型 第5.3节；选择操作符-> 第7.4.2节；结构类型 第5.6节。

8.7 switch语句

switch语句是一种基于控制表达式的值的多路分支。在实际使用中，它类似Pascal或Ada的“case”语句，但它的实现方式更接近Fortran的“computed goto”语句：

switch语句：

```
switch (表达式) 语句
```

case标签：

```
case 常量表达式
```

default标签：

```
default
```

关键字switch后面的控制表达式必须是整数类型，并且会执行寻常单目转换。关键字case后面的表达式必须是整型常量表达式（第7.11.2节）。嵌入在switch语句内部的语句有时称为switch语句体，它通常是条复合语句，但并不一定需要是复合语句。

case标签或default标签被认为属于包含它的最内层switch语句。switch语句体内的任何语句（或语句体本身）都可以加上case或default标签。事实上，同一条语句也可以加上几个case标签和一个default标签。case或default标签不允许出现在swicth语句体之外的其他地方，并且属于同一条switch语句的两个case标签不能具有相同值的常量表达式。任何switch语句最多只能拥有1个default标签。switch语句是按下面的方式执行的：

1. 控制表达式被求值。

2. 如果控制表达式的值与属于这条swicth语句的某个case标签的常量表达式的值相同，程序的控制就转移到这个case标签所指示的位置，就像goto语句一样。

3. 如果控制表达式的值不等于任何case标签，但是这条switch语句存在一个default标签，程序的控制就转移到这个default标签所提示的位置。

4. 如果控制表达式的值不等于任何case标签，并且不存在default标签，这条switch语句体内的任何语句都不会被执行。程序的控制转移到switch语句之后的地方。

在比较控制表达式和case表达式时，case表达式被转换为控制表达式的类型（在寻常单目转换之后）。

控制表达式与每个case表达式的比较顺序并没有定义，并且比较方式的实现取决于case表达式的数量和值。程序员常常假设switch语句被实现为一系列与case表达式相同顺序的if语句，但这并不一定正确。

当控制转移到一个case或default标签时，它将继续执行接下来的语句，忽略它所遇到的其他任何case或default标签，直到switch语句结束或者程序的控制被goto、return、break或continue语

句转移出switch语句。

尽管标准C允许任何整数类型的控制表达式，有些旧式的编译器并不允许它是long或unsigned long类型。标准C还允许编译器限制switch语句中可以出现的case标签的数量。这个限制在C89中为257，在C99中为1023，足以处理典型的char类型（8位）的所有值。

在C99中，如果任何可变修改类型的对象在任何case或default标签中可见，则这个对象的作用域必须覆盖整条switch语句。也就是说，可变修改类型的对象的作用域不能仅覆盖switch语句的一部分，除非这个作用域完全包含于一个case或default标签中。换句话说，我们无法把一个case或default标签“埋”在一个包含可变修改类型的对象的代码块中。

参考：break和continue语句 第8.8节；常量表达式 第7.11节；goto语句 第8.10节；整数类型 第5.1节；标签语句 第8.3节；return语句 第8.9节；可变修改类型 第5.4.5节。

使用switch语句

一般而言，switch语句体是条复合语句，它内部的顶层语句具有case和（或）default标签。应该注意到，case和default标签并不会更改程序的控制流。程序的执行并不会受到这些标签的影响。switch语句体内部可以使用break语句，终止switch语句的执行。

例子

```
switch (x) {
   case 1: printf("*");
   case 2: printf("***");
   case 3: printf("****");
   case 4: printf("*****");
}
```

在前面的switch语句中，如果x的值是2，则打印9个星号。之所以出现这个结果的原因是switch语句把控制转移到表达式为2的case标签，这时将执行参数为“**”的printf调用。接着，参数为“***”的printf调用被执行。最后，参数为“****”的printf调用被执行。如果想在每个case后面的printf调用执行之后终止switch语句体的执行，应该使用break语句：

```
switch (x) {
   case 1: printf("*");
           break;
   case 2: printf("***");
           break;
   case 3: printf("****");
           break;
   case 4: printf("*****");
           break;
}
```

在这个例子中，尽管最后一条break语句在逻辑上是不必要的，但是添加这条语句却是一种很好的编程风格。如果以后在这条switch语句中添加了第5个case标签，这种做法就可以防止出现程序错误。

在使用switch语句时，我们推荐坚持使用一种简单的风格：switch语句体始终应该是一条复合语句，所有属于这条switch语句的标签都应该出现在这条复合语句的顶层（这种风格也适用于goto语句）。而且，每个case（或default）标签（除了第1个之外）的前面必须出现两样东西之一：

或者是一条终止前一个case代码的break语句，或者是一条注释，注明确实需要从前一个case贯行执行到这个case。

尽管它被认为是一种良好的风格，但语言的定义并没有要求switch语句体是条复合语句，也没有要求case和default标签只能出现在复合语句的“顶层”，也没有要求case和default标签以任何顺序出现，或者必须出现在不同的语句中。

例子

在下面的代码段中，注释提醒读者fatal这个case后面缺少break语句是有意的。

```
...
case fatal:
   printf("Fatal ");
   /* Drops through. */
case error:
   printf("Error");
   ++error_count;
   break;
...
```

例子

下面是一个良好的意图可能会导致混乱的例子。下面这个程序片段的意图是尽可能提高效率：

```
if (prime(x)) process_prime(x);
else process_composite(x);
```

如果prime函数的实参是个质数，它返回1，如果它的参数是个合数，它返回0。程序的测试表明绝大多数prime调用都是针对较小的整数进行的。为了避免prime调用的开销，这段代码改用一条switch语句处理小整数，让default标签处理大整数。对代码进行了压缩之后，产生了下面的结果：

```
switch (x)
   default:
   if (prime(x))
      case 2: case 3: case 5: case 7:
         process_prime(x);
   else
      case 4: case 6: case 8: case 9: case 10:
         process_composite(x);
```

坦率地说，这是我们所见过的最奇怪的switch语句，但它仍然表明了自己的意图。

8.8 break和continue语句

break和continue语句用于更改循环内部的控制流（break语句还可以更改switch语句的控制流）。使用这两种语句可以实现与goto语句相同的功能，并被认为是更好的编程风格。

break语句：

```
break;
```

continue语句：

```
continue;
```

break语句的执行导致最接近的外层while、do、for或switch语句终止。程序的执行立即转移

到被终止的语句之后的位置。如果break语句并不位于任何迭代语句或switch语句的内部，将会出现错误。

continue语句终止最接近的外层while、do、for或switch语句体的执行。程序的执行立即转移到语句体的最后，受影响的迭代语句的执行从循环测试（如果是for语句，则是增值表达式）的求值重新开始。如果continue语句并不位于任何迭代语句的内部，将会出现错误。

和break语句不同，continue语句不会影响switch语句。continue语句可以出现在switch语句中，但它只影响最接近的外层迭代语句，而不是switch语句。

例子

break和continue语句可以根据goto语句来解释。考虑受break或continue语句所影响的语句：

```
while ( expression ) statement
do statement while ( expression );
for (expression1; expression2; expression3) statement
switch ( expression ) statement
```

想象一下，所有这些语句可以按照下面的方式进行改写：

```
{ while (expression) {statement C:;} B:;}
{ do {statement C:;} while (expression); B:;}
{ for (expression1; expression2; expression3) {statement C:;} B:;}
{ switch (expression) statement B:; }
```

在每个例子中，B和C都是标签，不会出现在外层函数的其他地方。这样，这些语句体中所出现的任何break语句相当于“goto B;”，这些语句（除了不允许的switch语句之外）体中所出现的任何continue语句相当于“goto C;”当然，它假设循环体中并没有其他包含break或continue语句的循环。

例子

break语句经常在两种重要的上下文环境中使用：终止一条switch语句中一个特定case的处理；永久地终止一个循环。第1种用途出现在switch语句中，如第8.7节所述。下面这个例子说明了第2种用途，它用输入字符填充一个数组，当数组填满或者当输入用尽时停止填充：

```
#include <stdio.h>
static char array[100];
int i, c;
…
for (i = 0; i < 100; i++) {
    c = getchar();
    if (c == EOF) break;    /* Quit if end-of-file. */
    array[i] = c;
}
/* Now i is the actual number of characters read. */
```

注意break是如何用于处理异常情况的。一般而言，在循环测试中处理正常情况是更好的风格。

例子

下面这个例子是在一个“永久”循环使用break语句的例子。它的思路是尽可能有效地找到数组a（长度为N）中最小的元素。它假设这个数组可以被临时修改：

```
int temp = a[0];
register int smallest = a[0];
register int *ptr = &a[N];  /* just beyond end of a */
…
```

```
for (;;) {
   while (*--ptr > smallest) ;
   if (ptr == &a[0]) break;
   a[0] = smallest = *ptr;
}
a[0] = temp;
```

这个程序的要点在于大多数工作是由一个紧凑的while循环完成的。这个循环使用ptr指针反向扫描一个数组，跳过那些大于当前所找到的最小元素的元素（如果元素以随机的顺序出现，则在找到了一个较小的元素时，大多数元素将大于这个元素，因此将被跳过）。这个while循环不会跳到数组之前，因为当前所找到的最小元素存储在第1个数组元素中。当while循环完成时，如果扫描到了数组的前面，则break语句终止外层循环。否则smallest和a[0]就被更新，并将再次进入这个while循环。

例子

把上面的代码与下面这个更简单、更显而易见的方法进行比较：

```
register int smallest = a[0];
register int j;
…
for (j = 1; j < N; ++j)
   if (a[j] < smallest) smallest = a[j];
```

这个版本当然更容易理解。但是，在这个循环的每次迭代时，必须显式地检查(j<n)，以免越过数组的尾部。相反，前面那个例子较为智能，可以隐式地执行这个检查。在效率至关重要的场合，就更适合使用前面更为复杂的代码。在其他情况下，则应该使用更简单、更清晰的代码。

参考：do语句 第8.6.2节；for语句 第8.6.3节；goto语句 第8.10节；switch语句 第8.7节；while语句 第8.6.1节。

8.9 return语句

return语句用于终止当前函数，并可以返回一个值：

return语句：

return 表达式$_{opt}$;

return语句的执行导致当前函数终止执行。程序的控制转移到函数的调用者或者紧随函数调用点之后的位置。

如果return语句中未出现表达式，则在C99中，函数的返回类型必须是void，否则这样的语句就是非法的。C89允许在非void函数中省略return语句的表达式，但是如果程序期待从这个函数调用中获得一个值，其行为就是未定义的。

如果return语句出现了表达式，则函数的返回类型不能是void，否则这条语句就是非法的。返回表达式将进行转换，就像赋值给函数的返回类型一样。如果不能进行这样的转换，则return语句是非法的。

如果程序的控制在到达函数体的结束时没有遇到return语句，则它的效果相当于执行了一条没有表达式的return语句。如果函数的返回类型不是void，这种行为就是未定义的。

例子

许多程序员在return语句的表达式两边加上括号，尽管这样做并非必需。这很可能是由于switch、if、while后面的表达式需要加上括号，从而养成了这种习惯。

```
int twice(int x) { return (2*x); }
```

参考：丢弃值 第7.13节；函数调用 第7.4.3节；函数定义 第9.1节；函数返回类型的一致 第9.8节。

8.10 goto语句

goto语句可以把控制转移到函数中的任何地方：

goto语句：

```
goto 命名标签 ;
```

命名标签：

```
标识符
```

goto关键字后面的标识符必须与当前函数中某条语句的命名标签相同。执行goto语句使程序的控制立即转移到函数中该标签所指定的位置，接下来所执行的正是goto语句所指定标签所提示的语句。

参考：标签语句 第8.3节。

使用goto语句

C允许goto语句把程序的控制转移到函数中的任何其他语句，但有些类型的转移可能导致容易混淆的程序，有些转移可能会妨碍编译器的优化。由于这些原因，我们建议不要从if语句的外部把控制转移到它的“then”或“else”分支中，不要从“then”转移到“else”分支或者相反，不要从switch或迭代语句的外部转移到switch或迭代语句体中，也不要从复合语句的外部转移到复合语句中。这种转移不仅应该在使用goto语句时避免，在switch语句中使用case和default标签时也应该避免这种转移。从复合语句的外部转移到复合语句的中间，将会绕过在复合语句的起始位置所声明的所有变量的初始化。作为一种良好的编程风格，应该尽量用break、continue和return语句代替goto语句。

例子

尽管存在这些警告，有时候goto语句仍然非常有用。下面这个例子在二维数组a中搜索v这个值。如果找到，就使用goto语句跳出双层嵌套的循环，并保留循环变量i和j的值。

```
#include <stdio.h>
int i, j, v, a[N][M];
…
for (i=0; i++; i<N)
  for (j=0; j++; j<M)
    if (a[i][j] == v) goto found;
printf("a does not contain %d\n", v);
…
found:
  printf("a[%d][%d]==%d\n", i, j, v);
```

参考：break和continue语句 第8.8节；控制表达式 第8.1节；if语句 第8.5节；标签语句 第8.3

节；return语句 第8.9节；switch语句 第8.7节。

8.11 空 语 句

空语句简单地由一个分号组成：

空语句：

```
;
```

空语句主要用于两种情况。第1种情况是作为迭代语句（while、do或for）的空语句体。第2种情况在结束复合语句的右花括号之前需要一个语句标签的时候。标签不能直接出现在右花括号的前面，它的后面必须是一条语句。

例子

下面这个循环不需要循环体，因为控制表达式完成了所有的工作：

```
char *p;
…
while ( *p++ ); /* find the end of the string */
```

例子

标签L出现在一条空语句之前：

```
if (e) {
   …
   goto L; /* terminate this arm of the 'if' */
   …
   L:;}
else …
```

参考：do语句 第8.6.2节；for语句 第8.6.2节；标签语句 第8.3节；while语句 第8.6.1节。

8.12 C++兼容性

8.12.1 复合语句

C++不允许跳入复合语句时跳过具有初始化值的声明。

例子

```
goto L; /* Valid but unwise in C; invalid in C++ */
{
   int i = 10;
L:
   …
}
```

参考：跳入复合语句 第8.4.2节。

8.12.2 循环中的声明

C99允许在for循环的初始化分句中声明变量，它们的作用域结束于循环体的尾部。这和标准C++一致。有些早期版本的C++把这类变量的作用域扩展到循环之后，直到外层的语句或函数。

8.13 练 习

1. 重写下面的语句，不使用for、while或do语句。

(a) `for(n=A;n<B;n++) sum+=n;`

(b) `while(a<b) a++;`

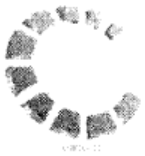
(c) `do sum+=*p; while (++p < q);`

2. 在下面这个程序片段的最后，j的值是什么？

```
{   int j=1;
    goto L;
    {
        static int i = 3;
    L:
        j = i;
    }
}
```

3. 在下面的程序段执行之后，sum的值是什么？

```
int i,sum = 0;
for(i=0;i<10;i++) {
    switch(i) {
    case 0: case 1: case 3: case 5: sum++;
    default: continue;
    case 4: break;
    }
    break;
}
```



第9章 函 数

本章讨论函数的用法，并描述了函数的声明和定义、形式参数和返回类型以及函数调用的细节。本书的前面已经出现了一些和函数有关的信息，例如第4.5.4节所描述的函数声明器，第5.8节所讨论的函数类型和函数声明。

和C的最初定义相比，函数的描述变得越来越复杂。标准C引入了一种新的函数声明方式（也是更佳的方式），也就是原型，它指定了和函数的形参有关的更详细信息。具有函数原型的函数被调用时所执行的操作与没有原型的函数不同。尽管原型和非原型形式的函数本身都很容易理解，但同一个函数混合使用这两种形式时，必须通过复杂的规则来确定应该执行什么操作（在C++中，必须使用函数原型）。

函数原型的存在与否是由函数声明器的语法所确定的（第4.5.4节）。简而言之，在传统C（即不使用原型时）中：

1. 函数的实参在调用之前将会自动提升（寻常实参转换）。
2. 不会对实参的类型或数量进行检查。
3. 任何函数都具有接受可变数量的参数的潜在能力。

与此相比，当使用原型时：

1. 函数的实参被转换，就像赋值为形式参数所声明的类型一样。
2. 实参的数量和类型必须与声明的类型匹配，否则程序就会出错。
3. 接受可变数量的实参的函数需要显式地指定，未指定的实参将执行默认的实参转换。

是否在C程序中使用原型，这是个令人烦恼的移植性问题。为了与非标准的编译器保持兼容，必须避免使用原型。为了与C++保持兼容，又必须使用原型。可以同时采用这两种形式，通过条件编译指令来决定需要使用哪种，但这种做法同样很笨拙。接下来的几节将同时讨论原型形式和非原型形式的函数声明，并讨论了一些移植性选项。

9.1 函数定义

函数定义引入了一个新函数，并提供了下面这些信息：

1. 函数所返回的值（如果有的话）的类型。
2. 形式参数的类型和数量。
3. 函数在定义它的文件之外的可见性。
4. 当函数被调用时所执行的代码。

下面显示了函数定义的语法。函数定义只能出现在C源文件或翻译单元（translation unit）的顶层。

翻译单元：

顶层声明
翻译单元　顶层声明

顶层声明：

声明
函数定义

函数定义：

函数定义指定符 复合语句

函数定义指定符：

声明指定符$_{opt}$ 声明器 声明列表$_{opt}$

声明列表：

声明
声明列表 声明

第4章讨论了其他顶层声明的语法。在C99之前，如果在函数定义的“声明指定符$_{opt}$”中未出现类型指定符，则默认为int类型。在C99中，必须要提供类型指定符。

在函数定义指定符中，声明器必须包含一个函数声明器，后者指定了函数标识符（就在左括号之前）。第4.4.4节显示了函数声明器的语法，为了方便读者，下面再次列出了它的语法：

函数声明器：

直接声明器（形参类型列表） (C89)
直接声明器（标识符列表$_{opt}$）

参数类型列表：

形参列表
形参列表，...

形参列表：

形参声明
形参列表，形参声明

形参声明：

声明指定符 声明器
声明指定符 抽象声明器$_{opt}$

标识符列表：

标识符
形参列表，标识符

如果对函数进行命名的函数声明器包含了形参类型列表，则这个函数定义被认为具有原型形式，否则它就是非原型形式（或传统形式）。在原型形式中，形参的名称和类型是在声明器中声明的，声明器后面的声明列表必须是空的。作为一种良好的风格，所有的C函数定义都应该写成原型形式。

在标准C之前的传统形式中，形参名是在声明器中列出的，但它们的类型是在声明器之后的声明列表$_{opt}$中指定的（以任意顺序）。所有的形参都应该在声明列表中声明，但在C99之前，省略形参声明就采用默认的int类型。C99把传统形式的函数定义当做一种过时的特性，意味着将来可能不对它提供支持。

例子

```
int f(int i, long j) { … }          /* prototype form */
int f(i,j) int i; long j; { … }     /* traditional form */
```

函数定义指示符的形式存在几个限制。函数定义所声明的标识符必须属于函数类型（由定义的声明器部分所指定）。也就是说，声明器必须包含一个函数声明器，指定了位于左括号之前的函数标识符。标识符不允许从typedef名称继承它的“函数性”。

函数的返回类型不能是数组或函数类型。

声明器必须指定函数的形参名称。如果声明器采用原型形式，形参声明必须包含一个声明器，而不是抽象声明器。如果声明器未采用原型形式，它必须包含一个标识符列表，除非这个函数不接受参数。为了避免标识符列表和形参类型列表之间的歧义，形参的名称不能与可见的typedef名称相同（这个限制在旧式的编译器中通常并不存在）。

形参声明中唯一允许使用的存储类别指定符是register。

声明列表$_{opt}$只能在非原型定义中使用，只能包含形参标识符的声明。有些传统C编译器允许其他声明（例如结构或typedef名称），但这些声明的含义存在问题，因此最好放在函数体中。

例子

为了说明这些规则，观察下面合法的函数定义：

定义	解释
void f() {...}	f是个不接受任何参数并且不返回任何值的函数（传统形式）
int g(x, y) int x, y;{...}	g是个接受2个整型参数，并返回1个整型结果的函数（传统形式）
int h(int x, int y){...}	h是个接受2个整型参数，并返回1个整型结果的函数（原型形式）
int (*f(int x))[]{...}	f是个接受1个整型参数，并返回1个指向整型数组的指针的函数（原型形式）

下面这些定义由于某种原因是非法的。假设typedef名称T被声明为“typedef int T();”。

定义	解释
int (*q)() {...}	q是个指针，而不是函数
T r {...}	r无法从typedef名称继承“函数性”
T s() {...}	s被声明为返回一个函数的函数
void t(int, double){...}	t的形参名称在声明器中未出现
void u(int x, y) int y;{...}	形参声明只有部分以原型形式出现

可以出现在函数定义中的存储类别指定符只有extern和static。extern表示这个函数可以被其他文件所引用，也就是说，这个函数的名称被导出到链接器。static指定符表示这个函数无法被其他文件所引用，也就是说，这个函数的名称并没有被导出到链接器。如果在函数定义中没有指定存储类别，则默认使用extern。在任何情况下，函数的可见范围是从定义点直到文件的末尾。特别需要说明的是，它在其他函数的函数体内可见。

参考：*声明器* 第4.5节；*extern存储类别* 第4.3节；*函数声明* 第5.8节；*初始化声明* 第4.1节；*static存储类别* 第4.3节；*类型指定符* 第4.4节。

9.2 函数原型

函数原型就是用原型语法（形参类型列表）所编写的函数声明，或者是用这种语法所编写

的函数定义。和传统函数声明一样，函数原型也声明了函数的返回类型。和传统函数声明不同，函数原型还声明了函数的形式参数的数量和类型。所有现代的C代码都应该使用原型来编写。C99把旧式的非原型形式当成是过时的。

取决于函数是不是不接受参数，或者具有固定数量的形参，或者具有可变数量的形参，原型一共可以分为3种类型：

1. 如果函数不接受任何参数，形参类型列表必须包含类型指定符void。在函数定义中，空的形参列表与void的含义相同，但这是一种过时的记法，应该避免使用。

例子

```
extern int random_generator(void);
static void do_nothing(void) { }   /* void is optional */
```

2. 如果函数接受固定数量的参数，形参列表指定了这些参数的类型。如果函数声明使用了原型形式，可以根据自己的喜好添加形参的名称（我们认为这种做法可以提高函数的可读性）。在函数定义中，必须提供形参的名称。

例子

```
double square(double x) { return x*x; }
extern char *strncpy(char *, const char *, size_t);
```

3. 如果函数接受可变数量或可变类型的形参，形参列表首先指定所有固定的形参，然后是个逗号，然后是个省略号（...）。形参列表中至少必须有1个固定形参，否则这个形参列表就无法被stdarg.h中的标准库函数所引用。

例子

下面是一个接受可变数量形参的函数声明。形参的名称使用了编译器所保留的标识符，就像标准库所要求的那样。

```
extern int fprintf( FILE *__file,
   const char *__format, ... );
```

例子

原型可以在任何函数声明器中使用，包括那些用于形成更复杂类型的声明器。signal函数（第19.6节）的标准C声明如下：

```
void (*signal(int sig, void (*func)(int siga)))(int siga);
```

它把signal声明为一个接受两个参数的函数：sig是个整数，func是个函数指针，它所指向的函数接受一个1个整型参数siga并且不返回任何值。signal函数返回一个指针，它与第2个参数具有相同的类型（它所指向的函数接受一个1个整型参数并且不返回任何值）。下面显示了signal的声明的一种更为清晰的形式：

```
typedef void sig_handler(int siga);
sig_handler *signal(int sig, sig_handler *func);
```

但是，在实际定义信号处理函数时，函数定义规则不允许使用sig_handler这个typedef名称。因此，必须重复这个类型：

```
void new_signal_handler(int sig a) {…}
```

在同一个声明中，在有些声明器中使用原型，但在另一些声明器中不使用原型也是可能的。如果我们把signal2声明为：

```
typedef void sig_handler2();  /* not a prototype */
sig_handler2 *signal2(int sig, sig_handler2 *func);
```

signal2函数的第2个参数将不采用原型形式，尽管signal2函数本身仍然采用了原型形式。

参考：函数声明器 第4.5.4节；函数声明 第5.8节；函数定义 第9.1节；void类型 第5.9节。

9.2.1 原型出现在什么时候

为了预测一个函数调用是如何执行的，重要的是程序员必须知道这个函数（或函数类型）被调用时是否受原型的支配。在下面这些情况下，函数调用将受到原型的支配：

1. 函数（或函数类型）的声明可见，并且这个声明采用了原型形式。
2. 函数定义可见，并且这个定义采用原型形式。

注意，它所要求的就是看到函数的任何原型，具有原型的函数可能还具有其他可见的非原型声明或定义。

如果同一个函数或函数类型具有两个或更多的原型声明，或者存在一个原型声明和一个原型定义，则这些声明和定义必须是兼容的（参见第5.11.4节的规则）。

参考：兼容和合成类型 第5.11节。

9.2.2 混合使用原型和非原型声明

尽管并不推荐在同一个函数中混合使用原型和非原型声明，标准C还是指定了一些条件，如果满足这些条件，这两种类型的声明是兼容的（参见第5.11.4节）。

如果在函数调用时向它提供的实参与函数定义不“匹配”，它的行为就是未定义的。在传统C中，程序员必须负责函数调用与定义匹配，语言所提供的帮助就是把实参和形参转换为一个更小、更容易管理的类型集合。在标准C中，通过使用原型声明，编译器可以在调用地点检查实参是否与原型匹配。

取决于函数声明所出现的位置，有可能有些函数受原型声明的支配，有些函数受传统声明的支配，还有一些函数受实际函数定义的支配。函数的调用和定义可能出现在同一个源文件中，也可能出现在多个不同的文件中。当有些调用不受原型的支配时，程序员必须承担额外的责任，确保这些调用中的实参与函数定义匹配。

例子

一般而言，可能存在多个不同的原型，它们各自与一个非原型声明兼容。例如，假设下面这个非原型声明

```
extern int f();
```

出现在一个C程序的某个位置。下面是一些与它兼容或不兼容的原型声明。

原型	是否与int f()兼容?	原因
extern double f(void)	否	形参列表没问题，但返回类型不兼容
extern int f(int, float)	否	float经过寻常实参转换后变成double。两种类型是不兼容的
extern int f(double x)	是	形参的类型在转换时不会发生变化

（续）

原型	是否与int f()兼容?	原因
extern int f(int i, ...)	否	原型不能包含省略号
extern int f(float *)	是	实参是个未转换的指针

一般情况下，只有1个原型与1个非原型函数定义匹配，这个原型有时称为这个函数的Miranda原型，因为它对应于一个函数定义，除此之外不存在其他的匹配。

在标准C中，接受可变数量参数的函数必须受原型的支配。这意味着所有接受可变数量参数的非标准C函数声明（例如printf）必须改写为使用原型，然后才能被标准C编译器所使用。

例子

例如，假设下面这个（非原型）定义出现在一个C程序中：

```
int f(x,y)
    float x;
    int y;
{…}
```

下面是一些与这个定义兼容或不兼容的原型声明。

原型	是否兼容	原因
extern double f(double x, int y)	否	形参列表没问题，但返回类型不兼容
extern int f(float, int)	否	第1个形参的类型必须是double
extern int f(float, int, ...)	否	原型不能包含省略号
extern int f(double a, int b)	是	这是唯一兼容的原型，形参的名称无关紧要

参考：兼容类型 第5.11节；printf 第15.11节。

9.2.3 合理地使用原型

根据原型所进行的实参检查并不保险。在一个由许多源文件组成的C程序中，编译器无法检查对一个函数的所有调用是否受同一个原型所支配、同一个函数的所有原型是否兼容以及所有的原型是否与函数定义匹配。

但是，如果程序员遵守下面这些简单的规则，就可以在实践中弥补上面这些漏洞：

1. 每个外部函数应该在一个头文件中有一个唯一的原型声明。通过使用唯一的原型，就可以消除同一个函数出现不兼容原型的可能性。

2. 调用这个函数的每个源文件应该包含提供了函数原型的那个头文件。这可以保证对这个函数的所有调用都受到同一个原型的支配，并允许编译器在调用地点对实参进行检查。

3. 包含函数定义的源文件也应该包含这个头文件。这允许编译器检查原型与声明是否匹配，并根据传递性，检查所有的调用是否与定义相匹配。

函数的定义并不一定要采用原型形式。

静态函数的使用也遵循相似的规则。要确保静态函数的原型形式的声明出现在对这个函数的任何调用以及该函数的定义之前。

9.2.4 原型和调用约定

本节主要针对编译器实现者，不过仍然可以向其他程序员提供一点和函数原型有关的观点。

函数原型的一个优点是它们允许编译器为有些函数生成更高效的调用序列。

例子

例如，按照传统C规则，即使一个函数被定义为接受一个float类型的形参，编译器仍然没有什么选择余地，必须把实参转换为double类型，接着调用这个函数，然后在函数内部把这个实参转换回float，再把它存储在形参中。在标准C中，如果编译器看到一个函数调用受下面这个原型所支配：

```
extern int f(float);
```

此时，编译器可以选择不把实参转换为double类型。另一方面，当它实现f的定义时，也可以作出同样的假设：

```
int f(float x) {…}
```

这里的微妙之处在于这个例子中，编译器并不需要与那些不受这个原型所支配的函数调用保持兼容，因为f的非原型声明（或定义）可能都不与这个原型兼容。因此，标准C并没有定义在没有可见的原型时调用f会发生什么情况。编译器可以选择把实参传递到寄存器中，即使非原型函数调用的约定是把所有的实参传递到堆栈中。

另一方面，如果一个原型声明可能是一个按照传统方式声明或定义的函数的Miranda原型，则编译器必须使用一种兼容的调用约定。

例子

如果函数g的调用受下面的声明之一所支配，它必须以一种兼容的方式实现：

```
extern short g();
extern short g(int,double);  /* Could be g's Miranda */
```

换句话说，如果一个标准C编译器看到下面这个函数调用：

```
process( a, b, c, d );
```

如果当时没有可见的原型，并且实际参数的类型如下时：

```
short a;
struct {int a,b;} b;
float *c;
float d;
```

则这个函数调用必须被实现为下面这个原型所规定的效果：

```
int process(int, struct {int a,b;}, float *, double);
```

这个规则实际上并没有建立一个将对以后的调用生效的原型。process的第2个调用应该出现在程序的后面或者出现在另一个源文件呢？这个时候，process的实参是3个double类型的值，因此第2个调用必须按照下面的原型实现，尽管这两个调用在执行时很可能是不兼容的。

```
int process( double, double, double );
```

下面对这些规则进行总结。只有当编译器看到一个函数调用受一个原型所支配，并且这个原型满足下面的条件时，编译器才允许使这个函数的所有调有都依赖这个原型：

1. 这个原型包含了一个与寻常实参转换不兼容的实参类型（char、short以及它们的无符号版本、float）。

2. 包含了一个省略号，表示可变数量的实参列表。

由于在大多数计算机上，char和short转换为int的开销很低，因此第1个规则主要用于float类型的实参。

第2个规则说明了编译器的标准调用约定不需要支持可变数量的实参列表（传统C必须支持）。例如，标准C编译器可以选择在它的标准约定中使用寄存器来保存任何函数的前4个（固定）参数，其余的参数则压入到堆栈中。这个约定在传统C中可能并不合适，因为有些接受可变参数的函数依赖所有的参数被连续传递到堆栈中。任何接受可变数量的参数的传统C函数（例如printf）必须进行重写，使它们采用原型形式，然后才能被标准C编译器所接受。

当存储类别register出现在原型声明中时，它就被忽略。这意味着register无法更改函数的调用约定，它只能在函数体中作为提示使用。

9.2.5 标准C和传统C的兼容性

标准C现已极为常用，所有的C程序都推荐使用原型。在一些较为少见的需要与不提供原型的编译器提供兼容的场合，可以通过不使用原型保持标准C和传统C的兼容。但是，这样做将放弃标准C编译器所带来的额外的类型检查的功能。解决这个问题的一种方法就是使用PARMS宏：

```
#ifdef __STDC__
#define PARMS(x) x
#else
#define PARMS(x) ()
#endif
```

然后，不再使用下面这样的原型声明：

```
extern int f(int a, double b, char c);
```

而是改用下面的声明（注意双层括号）：

```
extern int f PARMS((int a, double b, char c));
```

使用传统C编译器编译时，预处理器将把这一行展开为：

```
extern int f ();
```

但是标准C编译器将把它展开为：

```
extern int f (int a, double b, char c);
```

PARMS宏在函数定义中无法正确地工作，因此必须使用传统语法编写对应的函数定义，这也是标准C所能够接受的：

```
int f(a, b, c)
   int a; double b; long c;
{
   ...
}
```

在标准C中，传统定义并不会产生问题，只要在此之前源文件中出现了该函数的原型声明。

参考：__STDC__预处理宏 第3.3.4节。

9.3 形式参数声明

在函数定义中，形式参数是用原型语法或传统语法声明的。

在形参声明中，唯一可以出现的存储类别指定符是register，它提示编译器这个形参将被大量使用，最好在函数开始执行后存放在寄存器中。前面所讨论的关于哪些类型的形参可以标注为寄存器类型的常规限制也适用于这个场合（参见第4.3节）。

在标准C中，形式参数的作用域与在函数体的顶层所声明的标识符相同，因此它们无法在函数体内被隐藏或重新定义。当前有些C编译器允许这种重定义，但它几乎总是一种编程错误。

例子

考虑下面这个函数定义，如果它被一个遵循标准C的编译器所编译，double x;这个声明将是错误的。但是，有些非标准的编译器允许这种做法，从而使形参x在函数体内无法被访问。

```
int f(x)
    int x;
{
    double x;  /* hides parameter!? */
    …
}
```

在标准C中，形参可以被声明为除void之外的任何类型。但是，如果形参被声明为“返回T的函数”，它就会被隐式地改写为“指向返回T的函数的指针”。如果形参被声明为“T类型的数组”，它会被改写为“指向T类型的指针”。形参声明中的数组类型可以是不完整的。不管使用的是原型还是传统定义，都会进行这些调整，并且它们与调用地点所执行的默认参数转换（第6.3.5节）是同时进行的。在大多数情况下，程序员并不需要注意形参的这种变化，因为形参在函数内部是按照它们的声明类型使用的。

例子

假设FUNC函数被声明为

```
void FUNC(int f(void), int (*g)(void), int h[], int *j)
{
    int i;
    i = f();    /* OK */
    i = g();    /* OK */
    i = h[3];   /* OK */
    i = j[3];   /* OK */
    …
}
```

并且假设进行了下面这些FUNC调用：

```
extern int a(void), b[20];
…
FUNC( a, a, b, b );
```

在FUNC内部，表达式f将与g相同，h将与j相同。

有些标准之前的编译器拒绝“返回T的函数”类型的形参声明，要求它们显式地声明为“指向返回T的函数的指针”。

C99对数组形式参数的声明语法进行了扩展。在数组声明器顶层的方括号（[]）内部可以出现数组限定符列表。数组限定符（类型限定符）const、volatile和restrict支持数组和指针类型的相等关系。也就是说，下面这种形式的形参声明

```
T A[qualifier-list e]
```

被认为与下面的声明相同：

```
T * qualifier-list A
```

例子

假定有下面这些C99声明：

```
extern int f(int x[const 10]);
extern int g(const y[10]);
```

在函数f中，形参x被看成是具有int * const类型（指向int的常量指针），而函数g中的形参y被看成是具有const int *类型（指向常量int的指针）。

C99还允许在数组方括号中使用数组限定符static。它是向C编译器所提供的优化提示，表示实际的数组参数将是非空的，并且在进入函数时具有声明的长度和类型。如果没有这个限定符，作为实参传递给数组形参的可能是个null指针，使编译器难以知道在进入函数时预先提取形参数组的内容是否安全。

最后，对于原型中的C99形式数组参数声明（不是函数定义的一部分），长度可以用一个星号代替，表示实际参数将是一个可变长度的数组。在原型声明中，表示数组长度的任何非常量表达式都被看成与星号相同。函数定义必须提供一个非常量表达式来表示数组长度。

形式参数被看成是一个指定（或重写）类型的局部变量，并且复制了对应的实际参数传递给函数的值。对形参进行赋值是允许的，但这种赋值只会修改局部的参数值，而不是修改调用函数中的实际参数。根据重写规则，如果形参名被声明为函数或数组类型，它们就是左值，尽管属于这些类型的标识符在通常情况下并不是左值。

在传统C编译器中，在形参声明中包含typedef、结构、联合或枚举类型声明是允许的。在标准C中，形参声明中可以出现的名称只有形式参数的名称，并且它们都必须被定义（在C99之前，int类型的形参定义是可选的）。如果形参是使用原型语法声明的，则形参声明部分必须为空。

例子

```
int process_record(r)
    struct { int a; int b; } *r; /* not Standard C */
{
    ...
}
```

在传统C中，这通常是一种不良的编程风格。如果这些声明涉及形参，则这些声明应该移到函数的外面，使调用者也可以使用它们。如果这些声明不涉及形参，它们应该移到函数体内。

参考：数组限定符列表 第4.5.3节；枚举类型 第5.5节；函数声明器 第4.5.4节；函数原型 第9.2节；不完整类型 第5.4节；register存储类别 第4.3节；存储类别指定符 第4.3节；结构类型 第5.6节；typedef 第5.10节；联合类型 第5.7节；可变长度的数组 第5.4节；void类型 第5.9节。

9.4 形参类型的调整

本节只适用于未使用原型的标准C以及传统C。如果没有原型，必须执行一些函数实参值的转换（提升）。这些转换的目的是为了对函数的实参进行简化和规范化。它们称为寻常参数转换（或提升），第6.3.5节已经对它们有所说明。调用者应该预料到这种参数转换。在函数体执行之

前，C函数会负责把提升后的实参值转换为形参的声明类型。例如，如果函数F被声明为接受1个short类型的形参x，并且在调用F时指定了一个short类型的值，则执行这个调用时，就好象发生了下面这些事件：

1. 调用者对short类型的实参进行加宽，使它成为一个int类型的值。
2. 这个int类型的值被传递给F。
3. F把这个int值变窄为short类型。
4. F把这个short类型的值存储在形参x中。

幸运的是，这种转换的开销极小，至少对于整数是这样。受到转换影响的实参类型包括char、short、unsigned char、unsigned short和float。

例子

程序员应该注意有些标准C之前的编译器在进入函数时无法执行语言所要求的变窄转换。考虑下面这个函数，它具有char类型的参数：

```
int pass_through(c)
    char c;
{
    return c;
}
```

有些编译器在实现这个函数时就当它声明了int类型的形参：

```
int pass_through(c)
    int c;
{
    return c;
}
```

这种不正确的实现所产生的后果是实参的值并不会被变窄为char类型。也就是说，pass_through(0x1001)将返回0x1001而不是1。这个函数的正确实现应该像下面这样：

```
int pass_through(anonymous)
    int anonymous;
{
    char c = anonymous;
    return c;
}
```

参考：数组类型 第5.4节；浮点类型 第5.2节；函数实参转换 第6.3.5节；函数定义 第9.1节；函数原型 第9.2节；函数类型 第5.8节；整数类型 第5.1节；左值 第7.1节；指针类型 第5.3节。

9.5 参数传递约定

C只提供了参数的传值调用机制。这意味着从概念上说，实参的值将被复制到被调用函数的一个局部存储区域。例如，可以把形式参数放在赋值符的左边，但此时被修改的只是实参值的一份局部拷贝。如果程序员希望被调用函数更改它的实际参数，必须显式地传递实参的地址。

例子

下面的swap函数将无法正确地完成任务，因为x和y都是按值传递的：

```
void swap(x, y)
/* swap: exchange the values of x and y */
```

```
/* Incorrect version! */
    int x, y;
{
    int temp;
    temp = x; x = y; y = temp;
}
…
swap(a, b); /* Fails to swap a and b. */
```

这个函数的正确实现要求传递实参的地址：

```
void swap(x, y)
/* swap - exchange the values of *x and *y */
/* Correct version */
    int *x, *y;
{
    int temp;
    temp = *x; *x = *y; *y = temp;
}
…
swap(&a, &b); /* Swaps contents of a and b. */
```

形参的局部存储通常是通过压入堆栈实现的。但是，C语言并没有指定形参在堆栈中的压入顺序，也不禁止编译器把形参传递到寄存器中。在形式参数的名称前面使用取地址操作符&是合法的（除非它被声明为register存储类别），这就提示了在提取该形参的地址时，它必须存储在一个可取址的存储区域中（注意，形式参数的地址是实际参数的那份拷贝的地址，而不是实际参数本身的地址）。

在编写具有可变数量参数的函数时，程序员应该使用varargs或stdarg工具，以实现最大限度的移植性。

参考：取地址操作符& 第7.5.6节；函数原型 第9.2节；register存储类别 第4.3节；stdargs工具 第11.4节；varargs工具 第11.4.1节。

9.6 形参的一致

大多数现代编程语言（例如Pascal和Ada）会检查函数的形式参数和实际参数是否一致。也就是说，实参的数量以及每个实参的类型必须与形参一致。当函数是用原型声明时，标准C也会执行这种检查。

例子

在下面这个例子中，对sqrt的调用并不受原型的支配。因此，C编译器并不一定要警告程序员sqrt的实际参数类型是long，而形式参数被声明为double类型（事实上，如果函数的调用和定义位于不同的文件中，编译器也无法知道这种情况）。这个函数将简单地返回不正确的值：

```
double sqrt( x )      /* not a prototype */
    double x;
{
    …
}
long hypotenuse(x,y)
    long x,y;
{
    return sqrt(x*x + y*y);
}
```

当一个函数调用受标准C的原型所支配时，实际参数就会转换为对应的形式参数的类型。只有当无法进行这类转换时，或者实参的数量与形参的数量不对应时，C编译器才会拒绝这种程序。

例子

通过为上面的sqrt函数定义添加一个原型，这个例子就可以正确地工作：long类型的实参将被转换为double类型，而不需要程序员关注：

```
double sqrt( double x )     /* prototype */
{
   ...
}

long hypotenuse(x,y)
   long x,y;
{
   return sqrt(x*x + y*y);
}
```

作为一种良好的编程风格，我们推荐显式地把实参转换为预期的形参类型，除非这种转换只是重复寻常参数转换。也就是说，我们可以把上面这个例子的return语句改写为：

```
return  sqrt( (double) (x*x + y*y) );
```

有些C函数（例如fprintf）所接受的参数具有可变的数量和类型。在传统C中，varargs库工具提供了相当可靠的方式编写这种函数，尽管它是不可移植的，因为不同的编译器在varargs的形式上稍有区别。标准C使用了一种类似的机制stdarg，它还兼备可靠性和移植性。使有stdarg的函数在调用之前必须用原型声明，并使用省略号记法“,...”。这样，它就向编译器提供准备适当的调用机制的机会。

参考：实际参数的转换 第9.4节；函数实参的转换 第6.3.5节；函数原型 第9.2节；fprintf 第15.11节；

9.7 函数的返回类型

函数可以被定义为返回任何类型的值，除了“T类型的数组”和“返回T的函数”之外。这两种情况可以用指向数组或函数的指针来解决。和形式参数不同，返回类型不存在自动重写。

函数的返回值是由将导致函数终止的return语句中的表达式所指定的。第9.8节讨论了支配return表达式的规则。

函数的返回值并不是左值（返回的是“值”），因此函数调用无法出现在外层表达式中赋值符号的左边。

例子

```
f() = x;    /* Invalid */
*f() = x;   /* OK if f returns a pointer of suitable type */
f().a = x; /* Invalid--not an lvalue ( Section 7.4.2) */
```

参考：数组类型 第5.4节；函数调用 第7.4.3节；函数的形参 第9.4节；函数的类型 第5.8节；左值 第7.1节；指针类型 第5.3节；void类型 第5.9节。

9.8 返回类型的一致

如果一个函数的返回类型被声明为T（不是void），则出现在return语句中的表达式的类型必须可以通过赋值转换为类型T。事实上，在标准C和传统C中，都会进行这样的转换。

例子

在一个返回类型被声明为int的函数中，下面这条语句

```
return 23.1;
```

相当于

```
return (int) 23.1;
```

而后者又相当于

```
return 23;
```

如果一个函数的返回类型被声明为void，则在这个函数的return语句中提供一个表达式是错误的。在需要一个值的上下文环境中调用这种函数也是错误的。在旧式的并没有实现void的编译器中，在那些没有返回值的函数中省略类型指定符是常见的做法：

```
process_something()      /* probably returns nothing */
{
    …
}
```

为了提高程序的可读性，自己定义void类型也是可行的（第4.4.1节）。

如果函数的返回类型不是void，C89允许使用没有表达式的return语句，也就是简单的“return;”（C99禁止这种return语句，就像C++一样）。这个规则是为了与并没有实现void的编译器保持向后兼容。当一个具有非void返回类型并且使用了未提供表达式的return语句的函数执行时，实际所返回的值是未定义的。因此，在需要一个值的上下文环境中调用这样的函数是不明智的。

参考：*形式参数的调整* 第9.4节；*默认类型指定符* 第4.4.1节；*左值* 第7.1节；return语句 第8.9节；void类型 第5.9节。

9.9 main函数

所有的C程序必须定义一个称为main的外部函数。这个函数是程序的入口，也就是当程序启动时所执行的第1个函数。当这个函数返回时，程序也就终止，并且这个函数的返回值被看成是程序成功或失败的标志，就像调用了库函数exit一样。如果在到达main函数体的末尾时没有遇到返回语句，它就被看成是执行了return 0;语句。

标准C允许main函数被定义为接受0个或2个参数：

```
int main(void) { … }
int main() { … } /* also OK, but not recommended */
int main( int argc, char *argv[] ) { … }
```

如果未声明任何参数，就不会从环境向main函数传递任何信息，不过可以使用像getenv或system这样的库函数获取这类信息。

在C99之前，main函数的返回类型常常被省略，采用默认类型int。在C99中，这种做法不再被允许。

如果声明了参数，这些参数是由执行环境所创建的，而不是在C程序员的直接控制下。argc参数表示当用户或其他程序调用这个程序时传递给它的“程序参数”或“选项”的数量。argv参数是个指针向量，其中每个字符串指针分别表示传递给程序的参数。第1个字符串argv[0]是程序的名称。如果这个名称不可用，则argv[0][0]必须是'\0'。字符串argv[i]（i = 1、…、argc − 1）表示第i个程序参数。标准C要求argv[argc]是个null指针，但在有些旧式的编译器中却不是这样。argv向量以及它所指向的字符串必须是可以修改的，并且它们的值在程序执行期间不能被编译器或操作系统所修改。如果编译器并不允许大小写混合的字符串，则存储在argv中的字符串必须采用小写形式。

独立型C编译器以及一些软件框架（例如Microsoft Windows MFC）对C程序的启动可能采用一些特殊的约定。

例子

下面这个小程序打印出它的名称和参数。

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int i;
    printf("Name: %s\n", argv[0]);
    printf("Arguments: ");
    for( i=1; i<argc; i++)
        printf("%s ", argv[i]);
    printf("\n");
    return 0;
}
```

有些编译器允许向main函数提供第3个参数char * envp[]，它指向一个以null字符结尾的“环境值”向量，这个向量的每个元素都是一个以null字符结尾的字符串指针，形式为“name=value”。如果环境指针并不是main函数的参数，它可能采用全局变量的形式。有些UNIX系统使用全局变量environ保存环境指针。但是，使用标准C的getenv库函数访问环境信息可以使程序具有更好的移植性。

例子

假设envp保存了环境指针，下面的代码打印出环境的内容：

```
for(i=0; envp[i] != NULL; i++)
    printf("%s\n", envp[i]);
```

参考：exit 第16.5节；getenv 第16.6节；system 第16.7节。

9.10 内联函数

内联函数是C99新增的。如果函数的声明或定义中出现了函数指定符inline，这个函数就是内联函数。inline仅仅是向编译器所提供的提示，表示对内联函数的调用应该尽可能的快。这个名称来源于一种称为“内联展开”的编译器优化措施，对一个函数的调用将被该函数体的一份拷贝所代替。这就消除了函数调用的开销。许多C99之前的编译器对C进行了扩展，提供了内联

函数，并且C++也提供了这个功能。内联展开具有3个重要的原则：

1. 可见的定义。为了对一个函数调用进行内联展开，当编译器翻译这个调用时必须知道它的定义。在C99中，如果一个函数被声明为inline，则这个函数的定义必须在翻译单元中可见。

2. 自由的选择。编译器并不一定要执行内联展开。如果一个内联函数被调用了4次，只对2个调用进行内联展开而对另2个采用常规的函数调用也是完全可以的。C程序不能依赖于一个函数调用将被内联展开。

3. 相同的含义。当编译器对一个或多个调用进行内联展开时，它必须保证程序的行为就像这些函数被正常调用一样。内联展开只是一种优化，它并不会改变程序的含义。

任何静态函数都可以被指定为inline，因为对这类函数的所有调用以及该函数的定义都限于同一个翻译单元中。

外部函数则是另一回事，因为在通常的情况下，调用出现在一个翻译单元，而它的定义则位于另一个翻译单元。由于在任何内联声明的地方，函数的定义必须可见，因此外部函数的内联声明似乎只能出现在定义这个函数的翻译单元中。我们希望能够有一种方法让其他翻译单元“偷看”外部函数的定义，这样编译器就可以在这些翻译单元中对外部函数的调用进行内联展开。

这种“偷看”称为内联定义。如果一个函数在某个翻译单元中的所有顶层函数声明都包含了inline而没有包含extern，则这个函数在这个翻译单元中的定义称为内联定义（因此可以推断出必须存在一个这样的定义，它必须是inline而不是extern）。内联定义并没有为函数提供外部定义，因此必须在其他翻译单元中提供另一个外部定义。而且，内联定义是外部调用的一种替代方式，编译器可以使用这种替代方式执行内联展开。如果编译器选择不使用这种替代方式，它就产生一个常规的函数调用，把内联定义看成是个常规的extern声明。如果一个函数的所有内联声明和唯一的外部定义并不相同，则程序的行为是未定义的。使用内联定义的另一种方式是更改头文件，用内联声明替换函数声明。

例子

square函数返回它的参数的平方。square.h头文件向所有包含这个头文件的翻译单元提供了一个内联定义。如果编译器选择不对这个函数的调用进行内联展开或者需要取得这个函数的地址，这个内联定义也可以作为这个外部函数的声明。square.c翻译单元包含了这个内联定义，但同时提供了一个extern声明，使square.h中的定义成为外部函数定义。

```
// File: square.h
// Inline definition:
inline double square(double x) { return x*x; }

// File square.c
#include "square.h"
// Force an external definition using the inline code
extern inline square(double x);
```

一般而言，标准库头文件无法利用标准函数的inline定义，因为在有些情况下程序允许重新声明这些函数（宏）。但是，编译器可以使用它们自己的、不可移植的机制对标准库工具进行内联展开或者进行其他风格的特殊处理。

如果一个外部内联函数包含了一个静态对象的定义，就有可能出现问题。把内联定义中的静态对象与出现在另一个翻译单元的外部定义中的静态对象进行链接并不是一件容易的事情。

因此，C99禁止任何（非静态）内联函数定义一个可修改的静态对象，也禁止它包含对具有外部链接属性的标识符的引用。内联函数允许定义常量静态对象，但每个内联定义可能会创建它自己的对象。

参考：inline函数指定符 第4.3.3节。

9.11 C++兼容性

9.11.1 原型

为了与C++兼容，所有的函数都必须用原型声明。事实上，非原型形式在C++中具有不同的含义。在C++中，空的形参列表表示一个函数不接受任何参数，但在C中则表示接受未知数量的参数。

例子

```
int f(); /* Means int f(void) in C++, int f(...) in C */
int g(void); /* Means the same in both C and C++ */
…
x = f(2);    /* valid in C, not in C++ */
```

9.11.2 形参和返回类型中的类型声明

不要在形参列表或返回类型的声明中进行类型声明，C++不允许这种做法。

例子

```
struct s { … } f1(int i); /* OK in C, not in C++ */
void f2(enum e{…} x); /* OK in C, not in C++ */
```

9.11.3 返回类型的一致

在C++和C99中，对于非void返回类型的函数，必须返回一个适当类型的值。为了保持向后兼容，C89允许这种函数不返回任何值。

例子

```
int f(void)
{
   …
   return ; /* Valid but unpredictable in C;
               invalid in C++ */
}
```

参考：返回类型的一致 第9.8节。

9.11.4 main

在C++中，main函数不能被递归地调用，也不能取它的地址。C++对程序的启动施加了更严格的限制，因此编译器可能会把main函数当成一种特殊情况来处理。如果读者想要操纵main函数，可以简单地创建另一个函数，然后在main函数中调用这个函数，从而使它代替程序中的main函数。在C++中，如果控制到达main函数的尾部，将要执行隐式的“return 0;”。

9.11.5 内联

C99对函数的内联定义的规则并没有C++那样严厉，后者要求所有的内联定义和外部定义“完全”相同，而不仅仅是近似相等。C99允许有些翻译单元中的内联定义特殊化，并把保持相等的责任留给程序员。C++还要求内联函数在所有的翻译单元中都被声明为内联，而C99则无此要求。为了保持移植性，必须遵循C++的严格规则。

9.12 练 习

1. 下面哪些声明可以作为标准C的原型？

(a) `short f(void);`　　(d) `int f(i,j);`

(b) `int f();`　　(e) `int *f(float);`

(c) `double f(...);`　　(f) `int f(i) int i; {...}`

2. 下面显示了一些函数的声明和定义。哪几对是与标准C兼容的？

	声明	定义
(a)	`extern int f(short x);`	`int f(x) short x; {...}`
(b)	`extern int f();`	`int f(short x) {...}`
(c)	`extern f(short x);`	`int f(short int y) {...}`
(d)	`extern void f(int x);`	`void f(int x,...) {...}`
(e)	`extern f();`	`int f(x,y) short x,y; {...}`
(f)	`extern f();`	`f(void) {...}`

3. 下面显示了一些函数的声明和调用。在每个例子中，说明该调用在标准C中是否合法。如果合法，每个实际参数将应用什么约定。假设s的类型是short，ld的类型是long double。

	声明	调用
(a)	`extern int f(int *x);`	`f(&s)`
(b)	`extern int f();`	`f(s,ld)`
(c)	`extern f(short x);`	`f(ld)`
(d)	`extern void f(short,...);`	`f(s,s,ld)`
(e)	`int f(x) short x; {...}`	`f(s)`
(f)	`int f(x) short x; {...};`	`f(ld)`

4. 在下面这个程序片段中，P的调用是否受原型的支配？为什么？

```
extern void P(void);
…
int Q()
{
    extern P();
    P();
    …
}
```

5. 如果一个函数所声明的返回类型是short，下面哪些类型的表达式可以出现在该函数的return语句中，并在调用地点产生一个可预测的值？

(a) `int`

(b) `long double`

(c) `void`（例如，调用一个返回void的函数）

(d) `char *`

6. 解释square宏定义的哪些方面与第9.10节的内联版本不同：

```
#define square(x) ((x)*(x))
```

第二部分
C函数库



第10章　函数库简介

第11章　标准语言附加

第12章　字符处理

第13章　字符串处理

第14章　内存函数

第15章　输入/输出工具

第16章　基本工具

第17章　数学函数

第18章　时间和日期函数

第19章　控制函数

第20章　区域

第21章　扩展整数类型

第22章　浮点环境

第23章　复数运算

第24章　宽字符和多字节字符工具



第10章
函数库简介

标准C由C语言标准和一组标准函数库组成。这些函数库支持字符和字符串、输入和输出、数学函数、日期和时间转换、动态存储分配以及其他功能。每个函数库中的工具（类型、宏和函数）是由标准头文件所定义的。为了使用一个函数库中的工具，可以添加一条#include预处理器命令，引用表示该函数库的头文件。

例子

在下面的程序段中，头文件math.h允许这个程序访问余弦函数cos。

```
#include <math.h>
double x, y;
…
x = cos(y);
```

有些传统C编译器并没有通过头文件提供所有的库函数，因此程序员必须声明部分库函数。

对于那些被定义为函数的库工具，标准C允许编译器在真正的函数之外提供同名的类似函数的宏。宏可以提供简单函数的更快实现，或者可以用不同的名称调用一个函数。在使用宏时，需要注意它的每个参数表达式必须正好只求值1次，就像函数一样。如果确实需要访问真正的函数，而不管是否存在宏，必须像下面的例子一样绕过宏。

例子

假设我们担心有一个宏会隐藏math.h所提供的cos函数。下面这两种方法可以引用底层的函数。它们都依赖于可能的宏名后面不会立即出现左括号。这可以防止任何名称为cos的类似函数的宏被展开。

```
#include <math.h>
double a, b, (*p)(double);
…
p= &cos; a = (*p)(b); /* calls function cos, always */
a = (cos)(b);          /* calls function cos, always */
```

另外，也可以采取下面的方法简单地删除任何可能隐藏cos函数的宏：

```
#include <math.h>
#undef cos
…
a = cos(b);            /* calls function cos, always */
```

参考：#include 第3.4节；带参数的宏 第3.3.2节；#undef 第3.3.5节。

10.1 标准C工具

第10.3节对标准库工具进行了总结，列出了每个函数库头文件所定义的名称，以及描述这些工具的具体章节。如果读者想寻找一个特定的库工具名称，但并不知道它位于哪个头文件，可

以在本书最后的索引中查找。

在每个单独的章节中，每个工具是用标准C形式描述的。除非正文中另有说明，否则可以按照下面的方式对标准C定义进行修改，得到传统C形式的库函数定义。

1. 去除所有使用了标准C类型（例如long long或_Complex）的函数，或者是标准C（C89或C99）所新增的函数。

2. 去除限定符const、restrict和volatile。去掉出现在数组声明器方括号内部的static。

3. 把void *类型修改为char *。把size_t类型修改为int。

标准C中的库工具和头文件在许多方面具有特殊之处，主要是为了保护实现的完整性。

1. 函数库所使用的名称在原则上被保留。程序员无法定义与标准库的名称相同的外部对象。

2. 库头文件或文件名可以为编译器所“内建”，但是仍然必须包含头文件，才能使它们在程序中可见。也就是说，stdio.h实际上可能并不对应于一个称为“stdio.h”的#include文件。

3. 程序员可以按任意顺序包含库头文件任意多次（在传统C编译器中，可能并不是这样）。

例子

下面是库头文件所采用的一种典型机制，保证它不会被多次包含：

```
/* Header stddef.h */
#ifndef _STDDEF /* Don't try to redeclare */
#define _STDDEF 1
typedef int ptrdiff_t;
… /* other definitions */
#endif
```

保留的库标识符

除了第2.6节所列出的关键字之外，标准C保留了标准函数库所声明的标识符，供自己使用。另外，它还保留了一些可能由标准C编译器内部使用的其他标识符。一条容易记忆的规则是：不要把标准函数库所声明的任何标识符当做其他用途，并且不要使用以下划线开头的标识符。这应该可以避免程序转移到不同的标准C编译器时出现名称冲突。下面列出了更精确的规则。

标识符的类型	程序员可以用于
具有外部链接的库标识符（例如函数名errno）	在宿主型环境中，在任何时候都不能作为具有外部链接的名称被复用
具有文件作用域的库标识符以及库宏	如果包含了定义它们的库头文件，它们就无法作为具有文件作用域的名称或宏被复用
以下划线开头并且接下来是个大写字母或另一个下划线的标识符	无法用于任何目的，常常由C编译器的扩展所使用
以下划线开头的其他标识符	无法作为具有文件作用域的名称或标签

程序员无法自己编写标准库函数的替代版本。试图自己编写sqrt函数将会导致链接错误，因为这样就存在两个具有相同名称的函数。这个限制使C编译器在包装和内部使用标准库函数时具有更大的自由。

10.2 C++兼容性

C++语言包含了标准C运行时函数库，另外还添加了一些C++特有的库。这些额外的库均不

是以“.h”结尾，因此不会与C程序员所使用的库冲突。

C++在调用函数时采用一种不同的约定。一般而言，这意味着无法在C程序中调用C++函数。但是，C++提供了一种方式，以便在C++程序中调用C函数。它对C函数的声明具有两个要求：

1. 函数声明必须使用标准C的原型。C++要求使用原型。

2. 外部C声明必须显式地标记为具有C链接，方法是在C++声明的存储类别extern后面包含字符串“C”。

例子

如果在一个C函数中调用另一个C函数，可以采用下面的声明方式：

```
extern int f(void);
```

但是，如果这个函数是在一个C++函数中被调用的，它的声明必须是：

```
extern "C" int f(void);
```

如果需要在一个C++程序中声明一组C函数，可以同时为它们指定这个链接：

```
extern "C" {
    double sqrt(double x);
    int f(void);
    …
}
```

如果一个函数库可能同时被C或C++所调用，在编写它的头文件时，必须选择是在头文件中指定C链接，还是由包含这个头文件的C++程序提供链接声明。

例子

假设头文件library.h需要被C或C++程序所调用。第1种方法是通过__cplusplus宏（表示这是个C++程序），在头文件中包含extern "C"声明。

```
/* File library.h */
#ifdef __cplusplus
extern "C" {
#endif
…
/* C declarations */
…
#ifdef __cplusplus
}
#endif
```

第2种方法是使用常规的C声明编写头文件，并简单地要求C++用户在#include命令前面加上链接声明：

```
extern "C" {
#include "library.h"
}
```

在前面这个例子所使用的第2种方法中，它所调用的函数库在编写时肯定没有考虑C++。嵌套的extern "C"{}并不会有害处。

参考：__cplusplus宏 第3.9.1节。

10.3 库头文件和名称

10.3.1 assert.h

参见第19章。

```
assert    NDEBUG
```

10.3.2 complex.h

参见第23章。这个头文件是C99新增的。

```
cabs        catan       clog        csinf
cabsf       catanf      clogf       csinh
cabsl       catanh      clogl       csinhf
cacos       catanhf     complex     csinhl
cacosf      catanhl     _Complex_I  csinl
cacosh      catanl      conj        csqrt
cacoshf     ccos        conjf       csqrtf
cacoshl     ccosf       conjl       csqrtl
cacosl      ccosh       cpow        ctan
carg        ccoshf      cpowf       ctanf
cargf       ccoshl      cpowl       ctanh
cargl       ccosl       cproj       ctanhf
casin       cexp        cprojf      ctanhl
casinf      cexpf       cprojl      ctanl
casinh      cexpl       creal       CX_LIMITED_RANGE
casinhf     cimag       crealf      I
casinhl     cimagf      creall      imaginary
casinl      cimagl      csin        _Imaginary_I
```

10.3.3 ctype.h

参见第12章。

```
isalnum     isgraph     isupper
isalpha     islower     isxdigit
isblank     isprint     tolower
iscntrl     ispunct     toupper
isdigit     isspace
```

10.3.4 errno.h

参见第11章。

```
EDOM        ERANGE
EILSEQ      errno
```

10.3.5 fenv.h

参见第22章。这个头文件是C99新增的。

```
FE_ALL_EXCEPT    FE_TONEAREST     fegetround      fesetround
FE_DFL_ENV       FE_TOWARDZERO    feholdexcept    fetestexcept
FE_DIVBYZERO     FE_UNDERFLOW     FENV_ACCESS     feupdateenv
FE_DOWNWARD      FE_UPWARD        fenv_t          fexcept_t
FE_INEXACT       feclearexcept    feraiseexcept
```

```
FE_INVALID          fegetenv            fesetenv
FE_OVERFLOW         fegetexceptflag     fesetexceptflag
```

10.3.6 float.h

参见表5-3。

```
DBL_DIG             DBL_MIN_EXP         FLT_MAX_EXP         LDBL_MANT_DIG
DBL_EPSILON         DECIMAL_DIG         FLT_MIN             LDBL_MAX
DBL_MANT_DIG        FLT_DIG             FLT_MIN_10_EXP      LDBL_MAX_10_EXP
DBL_MAX             FLT_EPSILON         FLT_MIN_EXP         LDBL_MAX_EXP
DBL_MAX_10_EXP      FLT_EVAL_METHOD     FLT_RADIX           LDBL_MIN
DBL_MAX_EXP         FLT_MANT_DIG        FLT_ROUNDS          LDBL_MIN_10_EXP
DBL_MIN             FLT_MAX             LDBL_DIG            LDBL_MIN_EXP
DBL_MIN_10_EXP      FLT_MAX_10_EXP      LDBL_EPSILON
```

10.3.7 inttypes.h

参见第21章。这个头文件是C99新增的。

```
CNiLEASTN           PRIoMAX             PRIxPTR             SCNuFASTN
imaxabs             PRIoN               PRIXPTR             SCNuLEASTN
imaxdiv             PRIoPTR             SCNdFASTN           SCNuMAX
imaxdiv_t           PRIuFASTN           SCNdLEASTN          SCNuN
PRIdFASTN           PRIuLEASTN          SCNdMAX             SCNuPTR
PRIdLEASTN          PRIuMAX             SCNdN               SCNxFASTN
PRIdMAX             PRIuN               SCNdPTR             SCNxLEASTN
PRIdN               PRIuPTR             SCNiFASTN           SCNxMAX
PRIdPTR             PRIxFASTN           SCNiMAX             SCNxN
PRIiFASTN           PRIXFASTN           SCNiN               SCNxPTR
PRIiLEASTN          PRIxLEASTN          SCNiPTR             strtoimax
PRIiMAX             PRIXLEASTN          SCNoFASTN           strtoumax
PRIiN               PRIxMAX             SCNoLEASTN          wcstoimax
PRIiPTR             PRIXMAX             SCNoMAX             wcstoumax
PRIoFASTN           PRIxN               SCNoN
PRIoLEASTN          PRIXN               SCNoPTR
```

10.3.8 iso64.h

参见第11.5节。这个头文件是C89修正案1中新增的。

```
and                 bitor               not_eq              xor
and_eq              compl               or                  xor_eq
bitand              not                 or_eq
```

10.3.9 limits.h

参见表5-2。

```
CHAR_BIT            LLONG_MAX           SCHAR_MAX           UINT_MAX
CHAR_MAX            LLONG_MIN           SCHAR_MIN           ULLONG_MAX
CHAR_MIN            LONG_MAX            SHRT_MAX            ULONG_MAX
INT_MAX             LONG_MIN            SHRT_MIN            USHRT_MAX
INT_MIN             MB_LEN_MAX          UCHAR_MAX
```

10.3.10 locale.h

参见第20章。

LC_ALL	LC_MONETARY	lconv	setlocale
LC_COLLATE	LC_NUMERIC	localeconv	
LC_CTYPE	LC_TIME	NULL	

10.3.11 math.h

参见第17章。

acos	coshl	fmin	isinf
acosf	cosl	fminf	isless
acosh	double_t	fminl	islessequal
acoshf	erf	fmod	islessgreater
acoshl	erfc	fmodf	isnan
acosl	erfcf	fmodl	isnormal
asin	erfcl	FP_CONTRACT	isunorderedldex
asinf	erff	FP_FAST_FMA	p
asinh	erfl	FP_FAST_FMAF	ldexpf
asinhf	exp	FP_FAST_FMAL	ldexpl
asinhl	exp2	FP_ILOGB0	lgamma
asinl	exp2f	FP_ILOGBNAN	lgammaf
atan	exp2l	FP_INFINITE	lgammal
atan2	expf	FP_NAN	llrint
atan2f	expl	FP_NORMAL	llrintf
atan2l	expm1	FP_SUBNORMAL	llrintl
atanf	expm1f	FP_ZERO	llround
atanh	expm1l	fpclassify	llroundf
atanhf	fabs	frexp	llroundllog
atanhl	fabsf	frexpf	log10
atanl	fabsl	frexpl	log10f
cbrt	fdim	HUGE_VAL	log10l
cbrtf	fdimf	HUGE_VALF	log1p
cbrtl	fdiml	HUGE_VALL	log1pf
ceil	float_t	hypot	log1pl
ceilf	floor	hypotf	log2
ceill	floorf	hypotl	log2f
copysign	floorl	ilogb	log2l
copysignf	fma	ilogbf	logb
copysignl	fmaf	ilogbl	logbf
cos	fmal	INFINITY	logbl
cosf	fmax	isfinite	
cosh	fmaxf	isgreater	
coshf	fmaxl	isgreaterequal	
logf	nanl	remquol	sinhl
logl	nearbyint	rint	sinl
lrint	nearbyintf	rintf	sqrt
lrintf	nearbyintl	rintl	sqrtf
lrintl	nextafter	round	sqrtl
lround	nextafterf	roundf	tan
lroundf	nextafterl	roundl	tanf
lroundl	nexttoward	scalbln	tanh
MATH_ERREXCEPT	nexttowardf	scalblnf	tanhf
math_ errhandling	nexttowardl	scalblnl	tanhl
	pow	scalbn	tanl
MATH_ERRNO	powf	scalbnf	tgamma
modf	powl	scalbnl	tgammaf
modff	remainder	signbit	tgammal
modfl	remainderf	sin	trunc

```
NAN             remainderl      sinf            truncf
nan             remquo          sinh            truncl
nanf            remquof         sinhf
```

10.3.12 setjmp.h

参见第19.4节。

```
jmp_buf         longjmp         setjmp
```

10.3.13 signal.h

参见第19.6节。

```
raise           SIG_ERR         SIGFPE          signal
sig_atomic_t    SIG_IGN         SIGILL          SIGSEGV
SIG_DFL         SIGABRT         SIGINT          SIGTERM
```

10.3.14 stdarg.h

参见第11.4节。

```
va_arg          va_end          va_start
va_copy         va_list
```

10.3.15 stdbool.h

参见第11.3节。

```
bool                                    false
__bool_true_false_are_defined           true
```

10.3.16 stddef.h

参见第11.1节。

```
NULL            ptrdiff_t       wchar_t
offsetof        size_t
```

10.3.17 stdint.h

参见第21章。这个头文件是C99新增的。

```
INT_FASTN_MAX   INTN_C          SIG_ATOMIC_MIN  UINTN_MAX
INT_FASTN_MIN   INTN_MAX        SIZE_MAX        uintN_t
int_fastN_t     INTN_MIN        UINT_FASTN_MAX  UINTPTR_MAX
INT_LEASTN_MAX  intN_t          uint_fastN_t    uintptr_t
INT_LEASTN_MIN  INTPTR_MAX      UINT_LEASTN_MAX WCHAR_MAX
int_leastN_t    INTPTR_MIN      uint_leastN_t   WCHAR_MIN
INTMAX_C        intptr_t        UINTMAX_C       WINT_MAX
INTMAX_MAX      PTRDIFF_MAX     UINTMAX_MAX     WINT_MIN
INTMAX_MIN      PTRDIFF_MIN     uintmax_t
intmax_t        SIG_ATOMIC_MAX  UINTN_C
```

10.3.18 stdio.h

参见第15章。

```
BUFSIZ          fputs           printf          stderr
clearerr        fread           putc            stdin
EOF             freopen         putchar         stdout
fclose          fscanf          puts            TMP_MAX
feof            fseek           remove          tmpfile
ferror          fsetpos         rename          tmpnam
fflush          ftell           rewind          ungetc
fgetc           fwrite          scanf           vfprintf
fgetpos         getc            SEEK_CUR        vfscanf
fgets           getchar         SEEK_END        vprintf
FILE            gets            SEEK_SET        vscanf
FILENAME_MAX    _IOFBF          setbuf          vsnprintf
fopen           _IOLBF          setvbuf         vsprintf
FOPEN_MAX       _IONBF          size_t          vsscanf
fpos_t          L_tmpnam        snprintf
fprintf         NULL            sprintf
fputc           perror          sscanf
```

10.3.19 stdlib.h

参见第16章。

```
abort           _Exit           MB_CUR_MAX      strtof
abs             EXIT_FAILURE    mblen           strtol
atexit          EXIT_SUCCESS    mbstowcs        strtold
atof            free            mbtowc          strtoll
atoi            getenv          NULL            strtoul
atol            labs            qsort           strtoull
atoll           ldiv            rand            system
bsearch         ldiv_t          RAND_MAX        wchar_t
calloc          llabs           realloc         wcstombs
div             lldiv           size_t          wctomb
div_t           lldiv_t         srand
exit            malloc          strtod
```

10.3.20 string.h

参见第13章。

```
memchr          size_t          strcspn         strpbrk
memcmp          strcat          strerror        strrchr
memcpy          strchr          strlen          strspn
memmove         strcmp          strncat         strstr
memset          strcoll         strncmp         strtok
NULL            strcpy          strncpy         strxfrm
```

10.3.21 tgmath.h

参见第17.12节。这个头文件是C99新增的。

```
acos            cproj           hypot           nexttoward
acosh           creal           ilogb           pow
asin            erf             ldexp           remainder
asinh           erfc            lgamma          remquo
atan            exp             llrint          rint
atan2           exp2            llround         round
atanh           expm1           log             scalbln
carg            fabs            log10           scalbn
```

```
cbrt            fdim            log1p           sin
ceil            floor           log2            sinh
cimag           fma             logb            sqrt
conj            fmax            lrint           tan
copysign        fmin            lround          tanh
cos             fmod            nearbyint       tgamma
cosh            frexp           nextafter       trunc
```

10.3.22 time.h

参见第18章。

```
asctime         ctime           mktime          struct tm
clock           difftime        NULL            time
clock_t         gmtime          size_t          time_t
CLOCKS_PER_SEC  localtime       strftime
```

10.3.23 wchar.h

参见第24章。这个头文件是C89修正案1中新增的。

```
btowc           putwchar        wcschr          wcstok
fgetwc          size_t          wcscmp          wcstol
fgetws          swprintf        wcscoll         wcstold
fputwc          swscanf         wcscpy          wcstoll
fputws          tm              wcscspn         wcstoul
fwide           ungetwc         wcsftime        wcstoull
fwprintf        vfwprintf       wcslen          wcsxfrm
fwscanf         vfwscanf        wcsncat         wctob
getwc           vswprintf       wcsncmp         WEOF
getwchar        vswscanf        wcsncpy         wint_t
mbrlen          vwprintf        wcspbrk         wmemchr
mbrtowc         vwscanf         wcsrchr         wmemcmp
mbsinit         WCHAR_MAX       wcsrtombs       wmemcpy
mbsrtowcs       WCHAR_MIN       wcsspn          wmemmove
mbstate_t       wchar_t         wcsstr          wmemset
NULL            wcrtomb         wcstod          wprintf
putwc           wcscat          wcstof          wscanf
```

10.3.24 wctype.h

参见第24章。这个头文件是C89修正案1中新增的。

```
iswalnum        iswgraph        iswxdigit       wctype
iswalpha        iswlower        towctrans       wctype_t
iswblank        iswprint        towlower        WEOF
iswcntrl        iswpunct        towupper        wint_t
iswctype        iswspace        wctrans
iswdigit        iswupper        wctrans_t
```



第11章 标准语言附加



有些标准C函数库可以被看成是语言的一部分。它们提供了标准定义和参数化过程，使C程序具有更好的移植性。它们是独立型的编译器必须提供的，即便后者并未提供其他函数库。这些核心函数库由头文件float.h、iso646.h、limits.h、stdarg.h、stdbool.h、stddef.h和stdint.h组成。第5章描述了float.h和limits.h中的工具，第21章将描述stdint.h函数库，其他函数库将在本章中描述。

本章还描述了errno.h中的工具，尽管这个函数库并不被认为是语言附加的一部分。尽管名称看上去很像，但stdlib.h并不被认为是语言附加的一部分，它将在第16章中描述。

11.1 NULL、ptrdiff_t、size_t、offsetof

语法概要

```
#include <stddef.h>
#define NULL …
typedef … ptrdiff_t;
typedef … size_t;
typedef … wchar_t;
#define offsetof( type, member-designator )…
```

上面这些是stddef.h头文件所定义的工具。

NULL宏的值表示传统的null指针常量。许多编译器简单地把它定义为整数常量0或者转换为void *类型的0。在标准C中，为了方便起见，许多不同的头文件都定义了这个宏。

ptrdiff_t类型是一个由编译器所定义的有符号整数类型，它是两个指针相减的结果类型。大多数编译器使用long表示这种类型。size_t类型是一种无符号整数类型，它是sizeof操作符的结果类型。大多数编译器使用unsigned long表示这种类型。有时候，标准C之前的编译器使用（有符号）int类型表示size_t。在C99中，ptrdiff_t和size_t的最小值和最大值是在stdint.h中定义的。

随着处理器的速度越来越快，功能越来越强，内存也越来越大，难以用32位的指针来表示。C编译器可以使用C99的long long类型表示ptrdiff_t，用unsigned long long类型表示size_t。对于旧式的C代码，这可能会造成问题，因为它假定sizeof(size_t) = sizeof(ptrdiff_t) = sizeof(long)。

offsetof宏被展开为一个整型常量表达式（size_t类型），是member-designator在结构类型type中以字节计的偏移量。如果这个成员是个位段，其结果是不可预测的。如果offsetof并没有被定义（在非标准C的编译器中），常常可以按照下面的方式对它进行定义：

```
#define offsetof(type,memb) ((size_t)&((type *)0)->memb)
```

如果编译器并不允许以这种方式使用null指针常量，可以使用一个预字义的非null指针，并把该成员的地址减去结构的基地址来计算该成员的偏移值。

例子

在下面这个程序片段的最后，diff的值将会是1，size和offset的值将会相同（对于采用字节

编址并且sizeof(int)等于4的计算机，size和offset都将等于4）。

```
#include <stddef.h>
struct s {int a; int b; } x;
size_t size, offset;
ptrdiff_t diff;
…

diff = &x.b - &x.a;
size = sizeof(x.a);
offset = offsetof(struct s,b);
```

wchar_t类型是在stddef.h中定义的，但我们将在第24章讨论头文件wchar.h时再对它进行描述。

参考：整数到指针的转换 第6.2.7节；null指针 第5.3.2节；指针类型 第5.3节；sizeof操作符 第7.5.2节；stdint.h 第21章；指针的减法 第7.6.2节；wchar_t 第24.1节。

11.2 EDOM、ERANGE、EILSEQ、errno、strerror、perror

语法概要

```
#include <errno.h>
extern int errno;
or #define errno …
#define EDOM   …
#define ERANGE …
#define EILSEQ …
```

```
#include <stdio.h>
void perror(const char *s)
```

```
#include <string.h>
char *strerror(int errnum)
```

上面这些工具是在errno.h和其他头文件中定义的，它们支持标准函数库的错误报告功能。

外部变量errno用于保存库函数所产生的由编译器所定义的错误代码。按照传统，这些错误代码一般在errno.h头文件中被定义为宏，并且以E开头。所有的错误代码都是正整数，并且库函数不应该清除errno的值。在标准C中，errno并不需要是个变量，它也可以是个能够扩展为int类型的可修改左值的宏。

例子

errno可以按照下面这种方式进行定义：

```
extern int *_errno_func();
#define errno (*_errno_func())
```

例子

使用errno的一种典型方式是在调用一个库函数之前清除它的值，并在调用之后检查它的值：

```
errno = 0;
x = sqrt(y);
if (errno) {
   printf("?sqrt failed, code %d\n", errno);
   x = 0;
```

C编译器一般定义了可以存储在errno中的标准错误代码列表。errno.h所定义的标准代码有：

EDOM　实参并不位于数学函数可以接受的定义域之内。例如，向log函数提供一个负整数作为实参。

ERANGE　数学函数的结果超出了值域范围。这个函数具有一个良好定义的数学结果，但由于浮点数格式的限制而无法被表示。例如，试图使用pow函数计算一个很大的数的一个非常大的幂。

EILSEQ　在转换一个多字节字符序列时遇到了一个编码错误。这个错误最终是由mbrtowc或wcrtomb检测的，而它们又是被其他宽字符函数所调用的（C89修正案1）。

函数strerror返回一个指向错误信息字符串的指针，这个字符串的内容是由编译器所定义的，它不能被修改，但可能在以后调用strerror函数时被改写。

perror函数在标准错误输出流中打印下面这个序列：参数字符串s、一个冒号、一个空格、一条和errno中的当前错误代码有关的较短的错误信息和一个换行符。在标准C中，如果s是个null指针或者是个指向null字符的指针，则只打印错误信息。前缀字符串、冒号和空格并不会被打印。

例子

前面的sqrt例子可以按照下面的方式进行改写，使用perror函数：

```
#include <math.h>
#include <errno.h>
…
errno = 0;
x = sqrt(y);
if (errno) {
    perror("sqrt failed");
    x = 0;
}
```

如果调用sqrt失败，输出结果将如下所示：

```
sqrt failed: domain error
```

这并不是标准C的一部分。但是，在有些系统中，与errno的值对应的错误信息可能存储在一个字符串指针向量中，后者一般被称为sys_errlist，可以使用errno中的值进行检索。sys_nerr变量包含了可以用来对sys_errlist进行检索的最大整数。应该对这个值进行检查，以保证errno并没有包含非标准的错误代码。

参考：编码错误 第2.1.5节；mbrtowc 第11.7节；wcrtomb 第11.7节。

11.3 bool、false、true

语法概要

```
#include <stdbool.h>
#define bool _Bool
#define false 0
#define true 1
#define __bool_true_false_are_defined 1
```

stdbool.h头文件是C99新增的，它只包含了上面所显示的这些声明。这些表示布尔类型和值的名称与C++一致。

尽管在正常情况下并不允许在标准头文件中定义#undefine宏，但C99允许程序员使用#undefine，并可以根据自己的需要对bool、false和true宏进行重定义。

参考：_Bool类型 第5.1.5节。

11.4 va_list、va_start、va_arg、va_end

语法概要

```
#include <stdarg.h>
typedef ... va_list;
#define va_start( a_list ap, type LastFixedParm) ...
#define va_arg( va_list ap, type) ...
void va_end(va_list ap);
void va_copy(va_list dest, va_list src);
```

stdarg.h头文件中的工具向程序员提供了一种具有良好移植性的方式，访问可变数量的参数列表。像fprintf（隐式）和vfprintf（显式）这样的函数就需要可变数量的参数列表。

C一开始并没有对向函数传递实参的方式施加限制，结果程序员可能会根据计算机系统的行为作出一些不可移植的假设。最终，为了提高移植性，传统C形成了varargs.h工具。标准C采用了一个在stdarg.h中定义的类似工具。stdarg.h的用法和varagrs.h不同，因为标准C允许在实参列表的可变部分之前出现固定数量的参数，而在以前的实现中，整个实参列表都被当成可变部分。

下面列出了上面所定义的宏、函数和类型的含义。这些工具进行了风格化，尽可能减少对编译器的依赖：

va_list　这个类型用于声明一个局部状态变量（在这里统一称为ap），用于对参数进行遍历。

va_start　这个宏对状态变量ap进行初始化，并必须在任何va_arg或va_end调用之前被调用。在传统C中，va_start把ap中的内部指针设置为指向传递给函数的第1个实参。在标准C中，va_start接受一个额外的参数，也就是最后一个固定参数的名称，并把ap中的内部指针设置为指向传递给函数的第1个可变参数。

va_arg　这个宏返回实参列表中下一个参数的值，并把ap中的内部指针推进为指向下一个参数（如果有）。必须指定下一个参数的类型（经过寻常实参转换之后，通过type参数指定），使va_arg可以计算它在堆栈中的长度。在调用va_start之后第1次调用va_arg将返回第1个可变参数的值。

va_end　这个函数或宏应该在所有的参数都通过va_arg被读取之后再调用。它对ap和va_list执行必要的清理工作。

va_copy　（C99）这个宏在dest中复制src的当前状态，创建指向实参列表的第2个指针。然后，va_arg可以独立作用于src和dest。和src一样，对dest也必须调用va_end。

va_arg宏调用中的类型名称type必须采用这样的写法，在它后面添加一个“*”将产生“指向type的指针”类型。

C99新增的va_copy(saved_ap, ap)宏可以在使用va_arg(ap, type)在列表中向前遍历时，保留一个指向实参列表的指针。如有必要，可以使用va_arg(saved_ap, ap)回到前面的位置。

例子

我们说明如何在标准C中编写具有可变参数的函数。下一节将介绍传统C所采用的实现。printargs函数接受可变数量的不同类型的实参，并在标准输出中打印它们的值。printargs的第1个实参是个整型数组，提示了后面这些实参的数量和类型。这个数组由一个0元素表示结束。下面是printargs函数的用法一例。这个例子在传统C和标准C中都是合法的：

```
#include "printargs.h"
int arg_types[] = { INTARG, DBLARG, INTARG, DBLARG, 0 };
int main()
{
    printargs( &arg_types[0], 1, 2.0, 3, 4.0 );
    return 0;
}
```

printargs的声明以及整数类型指定符的值保存在printfargs.h文件中：

```
/* file printargs.h; Standard C */
#include <stdarg.h>
#define INTARG      1      /* codes used in argtypep[] */
#define DBLARG 2
...
void printargs(int *argtypep,  ... );
```

下面显示了标准C中printargs的对应定义。

```
#include <stdio.h>
#include "printargs.h"
void printargs( int *argtypep,  ... )     /* Standard C */
{   va_list ap; int argtype;
    va_start(ap, argtypep);

    while ( (argtype = *argtypep++) != 0 ) {
        switch (argtype) {
        case INTARG:
            printf("int: %d\n", va_arg(ap, int) );
            break;

        case DBLARG:
            printf("double: %f\n", va_arg(ap, double) );
            break;
        /*  ...  */
        }
    } /*while*/
    va_end(ap);
}
```

传统工具：varargs.h

传统C语法概要

```
#include <varargs.h>
#define va_alist ...
```

（续）

传统C语法概要

```
#define va_dcl ...
typedef ... va_list;
void va_start( va_list ap );
type va_arg( va_list ap, type);
```

在传统C中，可变参数是使用头文件varargs.h实现的。它具有2个新宏，并对va_start的定义作了修改：

va_alist　这个宏替换了接受可变数量参数的函数定义中的形参列表。

va_dcl　这个宏替换了函数定义中的形参声明。它的后面不能是分号，以允许它为空。

va_start　这个宏对状态变量ap进行初始化，并必须在任何va_arg或va_end调用之前调用。在传统C中，va_start把ap中的内部指针设置为指向传递给函数的第1个实参。它比标准C版本用到了更少的实参。

例子

下面是传统C中printfargs的声明。

```
/* file printargs.h; Traditional C */
#define INTARG      1      /* codes used in argtypep[] */
#define DBLARG 2
…
#include <varargs.h>
printargs( va_alist );
```

下面显示了printargs的传统C实现。唯一的区别是在函数实参列表以及在va_list的调用中。

```
#include <stdio.h>
#include "printargs.h"
printargs( va_alist          )/* Traditional C */
    va_dcl
{   va_list ap; int argtype, *argtypep;
    va_start(ap);

    argtypep = va_arg(ap, int *);
    while ( (argtype = *argtypep++) != 0 ) {
        switch (argtype) {
        case INTARG:
            printf("int: %d\n", va_arg(ap, int) );
            break;
        case DBLARG:
            printf("double: %f\n", va_arg(ap, double) );
            break;
        /* … */
        }
    }
    va_end(ap);
}
```

11.5 标准C的操作符宏

语法概要

```
#include <iso646.h>
#define and          &&
#define and_eq       &=
#define bitand       &
#define bitor        |
#define compl        ~
#define not          !
#define not_eq       !=
#define or           ||
#define or_eq        |=
#define xor          ^
#define xor_eq       ^=
```

C89修正案1添加了头文件iso646.h。这个头文件包含了一些宏定义，这些宏可以用于替换一些操作符标记。这些操作符可能难以在受限的源字符集（例如ISO 646）中表示。在C++中，这些标识符都是关键字。

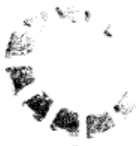
例子

在下面的3条if语句中，每条语句具有相同的效果。

```
#include <iso646.h>
...
if (*p || q != 0)      *p ^   = q;    /* customary */
if (*p ??!??| q != 0) *p ??' = q;    /* ISO C trigraphs */
if (*p or q != 0)      *p xor_eqq;    /* ISO C iso646.h*/
```

C89修正案1还提供了标点符号标记的重拼，例如用其他更常用的外国字母字符表示{和}。

参考：关键字 第2.6节；标记重拼 第2.4.1节；三字符组 第2.1.4节。



第12章
字符处理

有两种类型的工具可以用于对字符进行处理，一种是分类，另一种是转换。每个字符分类工具的名称以is开头，并且返回一个int类型的值。如果这个值不是零（真），表示实参字符属于指定的分类。如果这个值为零，表示实参字符不属于指定的分类。每个字符转换工具的名称以to开头，并返回一个int类型的值，表示一个字符或EOF。标准C保留了以is和to开头的名称，表示将来可能添加到标准函数库的分类和转换工具的名称。本章所描述的与字符相关的工具是在库头文件ctype.h中声明的。

C89修正案1定义了一组平行的分类和转换工具，用于处理宽字符。这些工具的名称以isw和tow开头，名称的剩余部分与基于字符的对应工具匹配。宽字符分类工具接受wint_t类型的实参，并返回一个int类型的真值。转换工具对wint_t类型之间的值进行映射。另外，还存在通用的分类函数wctrans和iswctrans以及通用的转换函数wctrans和towctrans，这是因为扩展字符集可能具有特殊的分类。这些工具都是在头文件wctype.h中定义的。

负整数EOF的值并不属于“真正字符”编码（WEOF的含义相同，但用于宽字符）。例如，fgetc（第15.6节）在遇到文件尾时返回EOF，因为此时已经没有可以读取的“真正字符”。必须记住，由于char类型在有些计算机上可能是有符号类型，因此如果出现了非标准的字符值，EOF可能无法与“真正的字符”进行区分（标准字符值总是非负值，即使char类型是有符号的）。这里所描述的所有工具都能正确地对char类型或unsigned char类型的所有可表示的值进行正确的操作，也能对EOF进行正确的操作。但是，除非另有所述，它们对其他整数值进行操作的结果是未定义的。WEOF用于wchar_t类型，就像EOF用于char类型一样，但是WEOF不一定是负值。

在标准C中，这些工具的形式考虑了需要支持一些区域的可能性。一般情况下，它尽可能减少对字符编码以及诸如“字母”这样的概念作出假设。这些函数的传统C版本大致与标准C的“C”区域形式对应，区别在于它还取消了所有ASCII依赖性（例如isascii和toascii）。

警告：有些非标准C编译器把char当做有符号类型，并支持unsigned char类型，但字符处理工具无法对unsigned char类型的所有值进行正确的操作。在有些情况下，这些工具甚至无法对char类型的所有值进行正确的操作，只能正确地处理“标准”字符值和EOF。

参考：EOF 第11.1节；WEOF 第11.1节；宽字符 第2.1.4节；wchar_t 第11.1节；wint_t 第11.1节。

12.1 isalnum、isalpha、iscntrl、iswalnum、iswalpha、iswcntrl

语法概要

```
#include <ctype.h>
int isalnum(int c);
int isalpha(int c);
int iscntrl(int c);
```

（续）

语法概要
`int isascii(int c);  /* 常见扩展 */`
`#include <wctype.h>` `int iswalnum(wint_t c);` `int iswalpha(wint_t c);` `int iswcntrl(wint_t c);`

isalnum函数测试c是否为字母数字字符。也就是说，在C区域中，是否为下面的字符之一：

```
0 1 2 3 4 5 6 7 8 9
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

按照定义，这个函数相当于：

```
isalpha(c) || isdigit(c)
```

isalpha函数测试c是否是个字母字符。也就是说，在C区域中，是否为下面的字符之一：

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

在任何区域中，当islower(c)或isupper(c)为真时，这个函数的结果也为真。当iscntrl(c)、isdigit(c)、ispunct(c)或isspace(c)任何一个为真的，这个函数的值为假。在其他情况下，它的值是由编译器所定义的。

iscntrl函数测试c是否为“控制字符”。如果当前使用的是标准的128字符的ASCII字符集，控制字符位于0到31（37_8或$1F_{16}$）以及127（177_8或$7F_{16}$）。isprint函数（第12.4节）是个增补函数，至少在标准ASCII实现中得到提供。

isascii函数并不是标准C的一部分，但它是C函数库的一个常用扩展。它测试c的值是否位于0到127（177_8或$7F_{16}$）的范围之内，这是标准的128字符的ASCII字符集的范围。和传统C的大多数字符分类函数不同，isascii可以对任何int类型的值执行正确的操作（在传统C中，它的参数也是int类型）。

在传统C中，这些函数接受char类型的参数，但返回int类型的结果。

例子

下面这个is_id函数当实参字符串S是个合法的C标识符时返回TRUE，否则返回FALSE。为了使这个函数能够正确地工作，当前区域必须是C。

```
#include <ctype.h>
#define TRUE 1
#define FALSE 0

int is_id(const char *s)
{
    char ch;
    if ((ch = *s++) == '\0') return FALSE; /*empty string*/
```

```
    if (!(isalpha(ch) || ch == '_')) return FALSE;
    while ((ch = *s++) != '\0') {
        if (!(isalnum(ch) || ch == '_')) return FALSE;
    }
    return TRUE;
}
```

宽字符工具

C89修正案1所定义的头文件wctype.h定义了3个额外的函数。

iswalnum函数相当于iswalpha(c) || iswdigit(c)。

iswalpha函数测试c是否为一个区域特定的宽字符“字母”。在任何区域中，当iswlower(c)或iswupper(c)为真时，这个函数的结果也为真。当iswcntrl(c)、iswdigit(c)、iswpunct(c)或iswspace(c)任何一个为真的，这个函数的值为假。在其他情况下，它的值是由编译器所定义的。

iswcntrl当c是区域特定的控制宽字符集的一个成员的码时返回非零值。无法打印的控制宽字符可以用iswprint（第12.4节）进行分类。

12.2 iscsym、iscsymf

非标准语法概要

```
#include <ctype.h>

int iscsym(char c);
int iscsymf(char c);
```

这两个函数在标准C中并不存在。iscsym函数测试c是否为可以在C标识符中出现的字符。iscsymf函数测试c是否可以作为标识符的第1个字符。

对于52个大小写字母以及下划线，iscsymf的结果为真。除此之外，iscsym对10个十进制数字的测试结果也为真。对于其他字符，这些函数的测试结果也可能为真，具体取决于编译器。

12.3 isdigit、isodigit、isxdigit、iswdigit、iswxdigit

语法概要

```
#include <ctype.h>

int isdigit(int c);
int isxdigit(int c)
```

```
#include <wctype.h>

int iswdigit(wint_t c);
int iswxdigit(wint_t c);
```

isdigit函数测试c是否是10个十进制数字之一。isxdigit函数测试c是否是22个十六进制数字之一：

```
0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f
```

在标准C之前，这些函数接受char类型的参数，但现在变为int类型。另外，读者可能会看到非标准的isodigit函数，它测试c是否为8个八进制数字之一。

宽字符工具

iswdigit函数（C89修正案1）测试c是否对应于十进制数字字符之一。iswxdigit函数测试c是否对应于十六进制字符之一。

12.4 isgraph、isprint、ispunct、iswgraph、iswprint、iswpunct

语法概要

```
#include <ctype.h>

int isgraph(int c);
int ispunct(int c);
int isprint(char c);
```

```
#include <wctype.h>

int iswgraph(wint_t c);
int iswpunct(wint_t c);
int iswprint(wint_t c);
```

isprint函数测试c是否为打印字符，即任何非控制字符。空格总是被看成是打印字符。isgraph函数测试c是否为“图形字符”之一，即除空格之外的其他打印字符。isprint和isgraph的区别仅在于它们如何处理空格字符。在大多数编译器中，isprint正好与iscntrl相反。但是在标准C中，并不是所有的区域都需要这样。在传统C中，这些函数接受char类型的参数，但它们返回int类型的值。

例子

如果当前使用的是128个字符的标准ASCII字符集，打印字符的范围位于040到0176，也就是空格加上下面这些字符：

```
! " # $ % & ' ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
[ \ ] ^ _
` a b c d e f g h i j k l m n o p q r s t u v w x y z
{ | } ~
```

图形字符与此相同，但不包括空格字符。

ispunct函数测试c是否为“标点字符”，也就是除空格和isalnum的测试结果为真之外的其他所有打印字符。

例子

如果当前使用的是128个字符的标准ASCII字符集，标点字符就是空格加上下面这些字符：

```
! " # $ % & ' ( ) * + , - . / : ; < = >
? @ [ \ ] ^ _ ` { | } ~
```

宽字符工具

iswprint函数（C89修正案1）测试c是否为宽打印字符，即在显示设备中至少占据1个位置的区域特定的宽字符，并且不是宽控制字符。

iswgraph函数相当于iswprint(c) && !iswspace(c)。iswpunct函数测试c是否为一个满足下面条

件的区域特定的宽字符：

```
iswprint(c) && !(iswalnum(c) || iswspace(c))
```

12.5 islower、isupper、iswlower、iswupper

语法概要

```
#include <ctype.h>
int islower(int c);
int isupper(int c);
```

```
#include <wctype.h>
int iswlower(wint_t c);
int iswupper(wint_t c);
```

在C区域中，islower函数测试c是否为26个小写字母之一，isupper函数测试c是否为26个大写字母之一。在其他区域中，它们可能返回其他字符，只要这些字符满足下面这些条件：

```
!iscntrl(c) && !isdigit(c) && !ispunct(c) && !isspace(c)
```

在传统C中，这些函数接受char类型的参数，但它们返回int类型的值。

宽字符工具

iswlower函数（C89修正案1）测试c是否对应于一个小写字母或满足下面条件的另一个区域特定的宽字符：

```
!iswcntrl(c) && !iswdigit(c) &&
!iswpunct(c) && !iswspace(c)
```

iswupper函数测试c是否对应于一个大写字母或满足与iswlower函数相同条件的另一个区域特定的字符。

12.6 isblank、isspace、iswhite、iswspace

语法概要

```
#include <ctype.h>
int isblank(int c);
int isspace(int c);
```

```
#include <wctype.h>
int iswspace(wint_t c);
```

isspace函数测试c是否为空白字符。在C区域中，isspace函数仅对制表符（'\t'）、回车符（'\r'）、换行符（'\n'）、垂直制表符（'\v'）、换页符（'\f'）和空格字符（' '）返回真。许多其他库工具使用isspace函数作用空白字符的定义。

isblank函数测试c是否为在一行文本中用于分隔单词的字符。它总是包括标准的空白字符空格（' '）和水平制表符（'\t'），并可能包含isspace测试结果为真的其他区域特定的字符。C区域不再包括其他空白字符。

有些C编译器提供了isspace的一种变型，称为iswhite。在传统C中，这些函数接受char类型

的参数，但它们返回int类型的值。

宽字符工具

iswspace函数（C89修正案1）测试c是否为满足下面条件的区域特定的宽字符：

```
!iswalnum(c) && !iswgraph(c) && !ispunct(c)
```

12.7 toascii

非标准语法概要

```
#include <ctype.h>
int toascii(int c);
```

非标准的toascii函数接受任何整数值，并把它缩减在合法的ASCII字符的范围之内（从0-127[177_8或$3F_{16}$]的代码），方法是丢弃最低7位之外的其他所有位。如果参数已经是个合法的ASCII字符，函数的返回结果与实参相同。

12.8 toint

非标准语法概要

```
#include <ctype.h>
int toint(char c);
```

非标准的toint函数返回一个十六进制数字的“权重”：0到9分别表示字符'0'到'9'，10到15分别表示字母'a'到'f'（或'A'到'F'）。如果实参不是个十六进制数字，这个函数的行为则依赖于编译器。

例子

这个工具在标准C并不存在，但可以很方便地实现。这个实现假定这些字符在标准编码方案中是连续的：

```
int toint( int c )
{
    if (c >= '0' && c <= '9') return c - '0';
    if (c >= 'A' && c <= 'F') return c - 'A' + 10;
    if (c >= 'a' && c <= 'f') return c - 'a' + 10;
    /* c is not a hexadecimal digit */
    return 0;
}
```

12.9 tolower、toupper、towlower、towupper

语法概要

```
#include <ctype.h>
int tolower(int c);
int toupper(int c);
```

```
#include <wctype.h>
wint_t towlower(wint_t c);
wint_t towupper(wint_t c);
```

如果c是个大写字母，则tolower函数返回对应的小写字母。如果c是个小写字母，toupper函数返回对应的大写字母。在其他情况下，就原封不动地返回实参。在有些区域中，大写字母可能不存在对应的小写字母（或反之）。在这些情况下，这两个函数也是原封不动地返回实参。

towlower和towupper函数是在C89修正案1中定义的。如果c是个宽字符且iswupper(c)为真，而d是个与c对应并且iswlower(d)为真的宽字符，则towlower(c)返回d并且towupper(d)返回c。否则，这两个函数就原封不动地返回它们的参数。

在使用非标准的编译器时，如果tolower函数的参数并不是大写字母或者toupper函数的参数并不是小写字母，必须注意它们的返回值。许多旧式的编译器只有当参数是个处于正确大小写状态的字母时才能正确地工作。允许向tolower和toupper传递更通用参数的编译器可能会提供这些函数的快速版本，即_tolower和_toupper宏。这两个宏所要求的参数具有更强的限制性，因此执行速度更快。非标准的函数签名是：

```
#include <ctype.h>
int tolower(char c);
int toupper(char c);
#define _tolower(c) …
#define _toupper(c) …
```

例子

如果读者所使用C函数库中的tolower函数无法对任意的参数都提供良好的行为，可以使用下面这个safe_tolower函数，它与tolower相似，但能够更安全地接受所有的参数。把safe_tolower编写为宏是困难的，因为参数将被多次求值（被isupper、islower和return语句所求值）：

```
#include <ctype.h>
int safe_tolower(int c)
{
    if (isupper(c)) return tolower(c);
    else return c;
}
```

12.10 wctype_t、wctype、iswctype

语法概要

```
#include <wctype.h>
typedef … wctype_t;
wctype_t wctype(const char *property);
int iswctype(wint_t c, wctype_t desc);
```

wctype函数和iswctype函数是在C89修正案1中定义的。它们实现了一种可扩展的、区域特定的宽字符分类工具。

wctype_t必须是标量类型，它包含了表示区域特定的宽字符分类的值。wctype函数创建了一个wctype_t类型的值，表示一种宽字符分类。这个分类是由称为property的字符串所指定的，后者是当前区域的LC_CTYPE类别特定的值。所有的区域必须允许property是表12-1所列出的任何字符串名称。这张表还列出了这些字符串名称的含义。

表12-1 wctype的属性名称

property名称	指定了什么分类	property名称	指定了什么分类
"alnum"	iswalnum(c)为真	"print"	iswprint(c)为真
"alpha"	iswalpha(c)为真	"punct"	iswpunct(c)为真
"cntrl"	iswcntrl(c)为真	"space"	iswspace(c)为真
"digit"	iswdigit(c)为真	"upper"	iswupper(c)为真
"graph"	iswgraph(c)为真	"xdigit"	iswxdigit(c)为真
"lower"	iswlower(c)为真		

iswctype函数测试c是否是desc这个值所表示的分类的成员。当iswctype被调用时LC_CTYPE类别的设置必须与当desc是由wctype所决定时LC_CTYPE的设置相同。

例子

对于任何宽字符c以及任何区域设置，表达式iswctype(c,wctype("alnum"))与iswalnum(c)具有相同的真值。其他属性字符串以及与它们对应的分类函数也具有对应的关系。

参考：LC_CTYPE 第11.5节；区域 第11.5节。

12.11 wctrans_t、wctrans

语法概要

```
#include <wctype.h>
typedef ... wctrans_t;
wctrans_t wctrans( const char *property );
wint_t towctrans( wint_t c, wctrans_t desc );
```

本节所描述的这些工具是在C89修正案1中介绍的。它们实现了一种可扩展的区域特定的宽字符映射工具。

wctrans_t必须是一种标量类型，它保存了表示区域特定的宽字符映射的值。wctrans函数创建了一个wctrans_t类型的值，表示宽字符之间的一个映射。这个映射是由字符串名称property指定的，后者是当前区域的LC_TYPE类别特定的值。所有的区域都必须允许property为下面这两个字符串值之一。下面同时列出了这些字符串的含义。

property值	指定了与下面所执行的相同映射
"tolower"	towlower(c)
"toupper"	towupper(c)

注：属性名称与函数名称不同。

towctrans函数把c映射到desc所指定的另一个宽字符。当towctrans被调用时LC_TYPE类别的设置必须与当desc的值是由wctrans确定时LC_TYPE的设置相同。

例子

对于任何宽字符c以及任何区域设置，表达式iswctrans(c,wctrans("tolower"))与iswlower(c)具有相同的值。其他属性字符串以及与它们对应的映射函数也具有对应的关系。

参考：LC_TYPE 第11.5节；区域 第11.5节。



第13章
字符串处理

按照约定，C的字符串就是以null字符（'\0'）结尾的字符数组。编译器会自动在所有的字符串常量后面添加一个额外的null字符。但是，确保在字符数组中所创建的字符串以null字符结尾则是程序员的责任。这里，所有的字符串处理方式都假定字符串以null字符结尾。

字符串中的所有字符（不包括最后的null字符）一起构成了该字符串的内容。空字符串不包含字符，由指向null字符的指针所表示。注意，这和null字符指针（NULL）并不相同，后者是不指向任何字符的指针。

把字符复制到一个目标字符串时，常常不会对目标字符串是否溢出进行检查。确保内存中的目标区域足以容纳结果字符串（包括结尾的null字符）是程序员的责任。

本章所描述的大多数工具是在库头文件string.h中声明的。有些标准C的转换工具是在stdlib.h中声明的。在标准C中，不会被修改的字符串参数一般被声明为const char *类型而不是char *，表示字符串长度的整型参数或返回值一般为size_t类型，而不是int类型。

C89修改案1添加了一组宽字符串函数，它们与常规的字符串函数同时存在。区别在于宽字符串函数所接受的参数为wchar_t *类型而不是char *类型。宽字符串函数的名称是用前缀wcs代替对应的字符串函数的前缀str。宽字符串是以null宽字符结尾的。在比较宽字符串时，所比较的是wchar_t元素的整型值。程序并不会对宽字符进行解释，因此不可能出现编码错误。

其他字符串工具是由内存函数（第14章）、sprintf（第15.11节）和sscanf（第15.8节）所提供的。

参考：wchar_t 第11.1节；宽字符串 第2.1.4节。

13.1 strcat、strncat、wcscat、wcsncat

语法概要

```
#include <string.h>
char *strcat( char *dest, const char *src );
char *strncat( char *dest, const char *src, size_t n );
```

```
#include <wchr.h>
wchar_t *wcscat( wchar_t *dest, const wchar_t *src );
wchar_t *wcsncat( wchar_t *dest, const wchar_t *src, size_t n );
```

strcat函数把字符串src的内容追加到字符串dest的尾部。这个函数返回dest的值。用于结束dest的null字符（以及它后面可能出现的其他字符）被src的字符所覆盖，并在最后添加一个新的null字符。它复制src中的字符，直到遇见src的null字符。这个函数假设从dest开始的内存区域具有足够的空间容纳这两个字符串。

除了参数和结果的类型之外，wcscat与strcat相同。

例子

下面的语句把3个字符串追回到D的后面。最后，D包含了字符串"All for one"。

"All for one.":

```
#include <string.h>
char D[20];
…
D[0] = '\0';        /* Set string to empty */
strcat(D,"All ");
strcat(D,"for ");
strcat(D,"one.");
```

strncat函数把src中的n个字符追加到dest的尾部。如果在复制了n个字符之前就遇到了src的null字符，则这个null字符也被复制，然后就不再复制任何字符。如果在src的前n个字符串中并没有null字符，则前n个字符就被复制到dest，并在dest中追加一个用于结束字符串的null字符。也就是说，总共有n + 1个字符被写入。如果n的值为0或负，则调用strncat没有任何效果。这个函数总是返回dest。在传统C中，strncat的最后一个参数为int类型。

wcsncat与strncat相似，区别在于参数和结果的类型。

如果两个参数字符串在内存中重叠，这些函数的行为是未定义的。

13.2 strcmp、strncmp、wcscmp、wcsncmp

语法概要

```
#include <string.h>

int strcmp( const char *s1, const char *s2 );
int strncmp( const char *s1, const char *s2, size_t n );
```

```
#include <wchr.h>

int wcscmp( const wchar_t *s1, const wchar_t *s2 );
int wcsncmp( const wchar_t *s1, const wchar_t *s2, size_t n );
```

strcmp函数按照词典顺序比较字符串s1和s2的内容（两个字符串均以null字符结尾）。如果s1小于s2，这个函数返回一个小于0的int值。如果s1等于s2，它返回int值0。如果s1大于s2，它返回一个大于0的int值。

例子

如果只是为了检查两个字符串是否相等，可以对strcmp的返回值求反：

```
if (!strcmp(s1,s2)) printf("Strings are equal\n");
else printf("Strings are not equal\n");
```

如果两个字符串的内容相同，它们就是相等的。如果出现下面这两种情况之一，字符串s1在词典顺序下小于字符串s2：

1. 这两个字符串在部分字符的位置上是相同的，在第一个不同的字符位置上，s1的字符值小于s2的字符值。

2. 字符串s1的长度短于s2，s1的内容和s2从头部开始并且长度与s1相同的那个子字符串的内容相同。

wcscmp（C89修正案1）和strcmp相似，区别在于参数的类型。

strncmp函数与strcmp相似，区别在于它最多只比较字符串s1和s2（均以null字符结尾）的前n个字符。在比较字符串时，如果字符串所包含的字符小于n个，则整个字符串都将进行比较。否则，字符串就被当成是n个字符的长度。如果n是0或负值，则两个字符串都将当做空字符串，比较的结果是相等，因此这个函数返回0。在传统C中，参数n的类型是int。

wcsncmp与strncmp相似，区别在于参数的类型。

memcmp函数（第14.2节）提供了与strcmp类似的功能。strcoll函数（第13.10节）提供了区域特定的比较工具。

13.3 strcpy、strncpy、wcscpy、cscncpy

语法概要

```
#include <string.h>
char *strcpy( char *dest, const char *src );
char *strncpy( char *dest, const char *src, size_t n );
```

```
#include <wchar.h>
wchar_t *wcscpy( wchar_t *dest, const wchar_t *src );
wchar_t *wcsncpy( wchar_t *dest, const wchar_t *src, size_t n );
```

strcpy函数把字符串src的内容复制到dest，覆盖dest的旧内容。src的完整内容都被复制，另外加上一个结尾的null字符，即使src的长度大于dest。这个函数返回实参dest。

wcscpy（C89修正案1）与strcpy相似，区别在于参数的类型。

例子

strcat函数（第13.1节）可以用strcpy和strlen（第13.4节）函数来实现，如下所示：

```
#include <string.h>
char *strcat(char *dest, const char *src)
{
    char *s = dest + strlen(dest);
    strcpy(s, src);
    return dest;
}
```

strncpy函数正好把n个字符复制到dest。它首先从src复制n个字符，如果src中null字符之前的字符少于n个，null字符就会作为填充字符写入到dest，直到凑足n个字符。如果src中有n个或更多的字符，则只有n个字符被复制，因此只有src的一份被截断的拷贝被复制到dest。只有当src的长度（不包含结尾的null字符）小于n时，strncpy在dest中所复制的那份拷贝才是以null结尾的。如果n的值是0或负，则调用strncpy没有任何效果。这个函数总是返回dest的值。在传统C中，参数n的类型是int。

wcsncpy（C89修正案1）与strcpy相似，区别在于实参的类型。

memcpy和memccpy（第14.3节）提供了与strcpy相似的功能。如果两个字符串在内存中重叠，strcpy、strncpy（以及它们的宽字符版本）的结果都是不可预料的。标准C所提供的memmove和wmemmove（第14.3节）就是为了处理可能出现的重叠情况。

13.4 strlen、wcslen

语法概要

```
#include <string.h>
size_t strlen(const char *s);
```

```
#include <wchar.h>
size_t wcslen(const wchar_t *s);
```

strlen函数返回s中结尾的null字符之前的字符数量。空字符串以null字符作为它的第一个字符，因此它的长度是0。在有些旧式的C编译器中，这个函数称为lenstr。

wcslen（C89修正案1）与strlen相似，区别在于它的参数类型。

13.5 strchr、strrchr、wcschr、wcsrchr

语法概要

```
#include <string.h>
char *strchr( const char *s, int c );
char *strrchr( const char *s, int c );
```

```
#include <wchar.h>
wchar_t *wcschr( const wchar_t *s, wchar_t c );
wchar_t *wcsrchr( const wchar_t *s, wchar_t c );
```

本节所讨论的函数都是在一个以null字符结尾的字符串中搜索单个字符c。在标准C函数中，s的结尾null字符被认为是该字符串的一部分。也就是说，如果c是个null字符（0），这个函数将返回s的结尾null字符的位置。在标准C中，参数c的类型是int。在传统C中，它的类型是char。这些函数的返回值是个指向非const值的指针。但是，在实际中，如果第一个参数所指向的是个const对象，它所指定的对象也将是const性质的。在这种情况下，试图往函数所返回的指针所指向的对象中存储一个值将导致未定义的行为。

strchr函数在字符串s中搜索字符c的第一次出现。如果在这个字符串中找到了字符c，这个函数就返回指向它所找到的第一个c的指针。如果未找到这个字符，这个函数就返回一个null指针。

wcschr函数（C89修正案1）与strchr函数相似，区别在于参数和返回值的类型。

strrchr函数与strchr函数相似，区别在于它返回指向字符c在字符串中的最后一次出现的指针。如果未找到该字符，这个函数返回一个null指针。

wcsrchr函数（C89修正案1）与strrchr相似，区别在于参数和返回值的类型。

传统C的strpos函数与strchr相似，区别在于返回值的类型是int，表示c的第一次出现所在的位置（s的第一个字符的位置被认为是0）。如果未找到该字符，这个函数返回-1。strrpos函数与strpos相似，区别在于它返回c的最后一次出现所在的位置。strpos和strrpos函数在标准C中均不存在。

memchr和wmemchr函数（第14.1节）提供了与strchr和wcschr相似的功能。在有些C编译器中，strchr和strrchr分别称为index和rindex。有些C编译器提供了scnstr函数，它是strpos的一种变型。

例子

下面这个how_many函数使用strchr函数计算一个指定的非空字符在一个字符串中所出现的次数。参数s将被反复更新，指向字符串中最后一次所找到的字符之后的那个位置。

```
int how_many(const char *s, int c)
{
    int n = 0;
    if (c == 0) return 0;
    while(s) {
        s = strchr(s, c);
        if (s) n++, s++;
    }
    return n;
}
```

13.6 strspn、strcspn、strpbrk、strrpbrk、wcsspn、wcscspn、wcspbrk

语法概要

```
#include <string.h>

size_t strspn( const char *s, const char *set );
size_t strcspn( const char *s, const char *set );
char *strpbrk( const char *s, const char *set );
```

```
#include <wchar.h>

size_t wcsspn( const wchar_t *s, const wchar_t *set );
size_t wcscspn( const wchar_t *s, const wchar_t *set );
wchar_t *wcspbrk( const wchar_t *s, const wchar_t *set );
```

本节所描述的函数都在一个以null字符结尾的字符串s中搜索一些字符，它们是根据这些字符是否包含在另一个以null字符结尾的字符串set中进行搜索的。第二个参数被看成是一组字符，至于它所包含的字符的顺序以及它们是否重复，则无关紧要。

strspn函数在字符串s中搜索set字符串没有包含的字符的第一次出现，它将跳过set所包含的字符。这个函数的返回值是s中由set中的字符所组成的最长起始片段的长度。如果s中的每个字符均在set中出现，则这个函数返回s的总长度（不包括结尾的null字符）。如果set是个空字符串，则s的第一个字符找不到，函数返回0。

strcspn函数与strspn相似，区别在于它在s中搜索set字符串中所包含的任何字符的第一次出现，跳过那些在set中不存在的字符。

strpbrk函数与strcspn相似，区别在于它返回指向第一个找到的set字符的指针，而不是被跳过的字符数量。如果set中的字符均未在s中找到，这个函数就返回一个null指针。

非标准函数strrpbrk与strpbrk具有相同的签名，但它返回指向s中所找到的最后一个set字符的指针。如果s中的所有字符均不在set中，这个函数就返回一个null指针。

wcsspn、wcscspn和wcspbrk函数（C89修正案1）与它们对应的str版本相似，区别在于参数和结果的类型。

在较为罕见的情况下，strspn和strcspn又称为notstr和instr。

例子

is_id函数判断输入字符串是否为合法的C标识符。它使用strspn函数观察输入字符串的所有

字符是否为字母、数字或下划线字符。如果是，就进行最后一个测试，判断第一个字符是否不是数字。读者可以把这个解决方案与第12.1节的例子进行比较。

```
#include <string.h>
#define TRUE (1)
#define FALSE (0)

int is_id(const char *s)
{
    static char *id_chars =
           "abcdefghijklmnopqrstuvwxyz"
           "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
           "0123456789_";
    if (s == NULL) return FALSE;
    if (strspn(s,id_chars) != strlen(s)) return FALSE;
    return !isdigit(*s);
}
```

13.7 strstr、strtok、wcsstr、wcstok

语法概要

```
#include <string.h>

char *strtok( char *str, const char *set );
char *strstr( const char *src, const char *sub );
```

```
#include <wchar.h>

wchar_t *wcstok(
  wchar_t *str, const wchar_t *set,wchar_t **ptr );
wchar_t *wcsstr(
  const wchar_t *src, const wchar_t *sub );
```

strstr函数是标准C新增的。它在字符串src中查找字符串sub的第一次出现，并返回指向这个被找到的子字符串的起始位置的指针。如果sub在src中不存在，这个函数就返回一个null指针。wcsstr函数（C89修正案1）与strstr相似，区别在于参数和结果的类型。

strtok函数使用set字符串中的字符把字符串str分隔为不同的标记。对于每个标记，都会调用strtok，并可能会修改后续调用中set的值。第一次调用包含了字符串str。在后续的调用中，传递一个null指针作为第一个参数，指示strtok函数从前一个标记的末尾继续开始工作（如果想使用strtok在字符串中查找更多的标记，源字符串str必须不能被修改）。

更精确地说，如果str并不是空字符串，则strtok首先跳过str中那些在set中也出现的字符。如果str中的所有字符均在set中出现，则strtok返回一个null指针，并把一个内部状态指针设置为null。在其他情况下，这个内部状态指针被设置为指向str中第一个并未出现在set中的字符，然后继续执行，就当str已经为空。

如果str和内部状态指针都为空，则strtok返回一个null指针，内部状态指针不会发生变化（这用于处理所有的标记都返回之后的额外strtok调用）。如果str为空，但内部状态指针不为空，则这个函数从内部状态指针的起始位置开始搜索第一个出现的set字符。如果找到了这样的字符，这个字符就用'\0'进行改写，并且strtok返回内部状态指针的值，然后内部状态指针被调整为指向新插入的空字符之后的位置。如果未找到字符，strtok就返回内部状态指针的值，并把内部状态指针设置为null。

标准C的库工具不能以程序员可以检测到的任何方式更改str的内部状态。也就是说，程序员不必担心使用strtok的库函数会与程序员自己所使用的strtok调用发生冲突。

wcstok函数（C89修正案1）与strtok相同，区别在于它的参数和结果的类型。另外，额外的ptr参数间接指定了一个指针，作为strtok函数的“内部状态指针”。也就是说，wcstok的调用者为内部状态提供了一个位置。

如果strstr或wcsstr的第一个参数是个指向常量字符串的指针，则返回值也将是这种类型的，尽管它并没有被声明为指向const的指针。

例子

下面这个程序从标准输入读取文本行，并使用strtok把这些文本行分割为“单词”，也就是由空格、逗号、句号、引号和问号所分隔的字符序列。这些单词将打印到标准输出中：

```
#include <stdio.h>
#include <string.h>
#define LINELENGTH 80
#define SEPCHARS " .,?\"\n"
int main(void)
{
    char line[LINELENGTH];
    char *word;

    while(1) {
        printf("\nNext line?  (empty line to quit)\n");
        fgets(line,LINELENGTH,stdin);
        if (strlen(line) <= 1) break;   /* exit program */
        printf("That line contains these words:\n");
        word = strtok(line,SEPCHARS);/* find first word */
        while (word != NULL) {
            printf("\"%s\"\n",word);
            word = strtok(NULL,SEPCHARS);/*find next word*/
        }
    }
}
```

下面是这个程序的一个示例执行：

```
Next line?  (empty line to quit)
"My goodness," she said, "Is that right?"
That line contains these words:
"My"
"goodness"
"she"
"said"
"Is"
"that"
"right"

Next line?  (empty line to quit)
```

13.8 strtod、strtof、strtold、strtol、strtoll、strtoul、strtoull

参见第16.4节。

13.9 atof、atoi、atol、atoll

参见第16.3节。

13.10 strcoll、strxfrm、wcscoll、wcsxfrm

语法概要

```
#include <string.h>
int strcoll( const char *s1, const char *s2 );
size_t strxfrm(
  char *dest, const char *src, size_t len );
```

```
#include <wchar.h>
int wcscoll(const wchar_t *s1, const wchar_t *s2);
size_t wcsxfrm(
  wchar_t *dest, const wchar_t *src, size_t len);
```

strcoll和strxfrm函数提供了区域特定的字符串排序工具。strcoll函数对字符串s1和s2进行比较，并根据s1是大于、等于还是小于s2，分别返回一个大于、等于或小于0的整数。它所执行的比较是根据区域特定的比较约定（使用setlocale所设置的LC_COLLATE，第11.5节）进行的。相比之下，strcmp和wcscmp函数（第13.2节）总是使用根据目标字符集（char或wchar_t）的常规比较序列对两个字符串进行比较。

wcscoll函数（C89修正案1）与strcoll相似，区别在于它的参数类型。

strxfrm函数把字符串src转换（用一种稍后描述的方式）为另一个字符串，存储在字符数组dest中。这个数组假定至少具有len个字符的长度。strxfrm函数返回存储这个字符串所需要的字符数量（不包括结尾的null字符）。因此，如果strxfrm的返回值大于或等于len，或者如果src和dest在内存中重叠，dest的最终内容是未定义的。另外，如果len是0并且dest是个null指针，strxfrm就简单地对应于src计算并返回转换字符串。

strxfrm函数在转换字符串时所使用的方式是转换之后的字符串可以使用strcmp函数决定正确的排序顺序。也就是说，如果s1和s2都是字符串，t1和t2是strxfrm函数根据s1和s2所产生的转换字符串，则：

- `strcmp(t1, t2) > 0` 如果 `strcoll(s1, s2) > 0`
- `strcmp(t1, t2) == 0` 如果 `strcoll(s1, s2) == 0`
- `strcmp(t1, t2) < 0` 如果 `strcoll(s1, s2) < 0`

wcsxfrm函数（C89修正案1）与strxfrm函数相似，区别在于它的参数类型。wcscmp函数必须用于对经过转换的宽字符串进行比较。

strcoll和strxfrm函数具有不同的效率权衡。strcoll函数并不要求程序员提供额外的存储空间，但它在每次被调用时都需要在内部执行字符串转换。如果必须对同一组字符串进行多次比较时，使用strxfrm的效率可能更高。

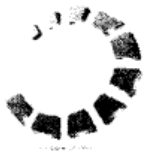
例子

下面这个transform函数使用了strxfrm，根据参数s创建一个转换字符串。这个字符串的空间是动态分配的：

```
#include <string.h>
#include <stdlib.h>

char *transform( char *s )
/* Return the result of applying strxfrm to s */
{
    char *dest; /* Buffer to hold transformed string */
    size_t length; /* Buffer length required */
    length = strxfrm(NULL,s,0) + 1;
    dest = (char *) malloc(length);
    strxfrm(dest,s,length);
    return dest;
}
```

第14章 内存函数



本章所描述的工具向程序员提供了高效的方式，对内存块进行复制、比较和设置。在标准C中，这些函数被认为是字符串函数的一部分，是在库头文件string.h中声明的。在旧式的编译器中，它们是在自己的头文件memory.h中声明的。

在标准C中，内存块是由void *类型的指针所指定的，在传统C中，这个类型是char *。在标准C中，内存被解释为unsigned char类型的对象数组。在传统C中，则不存在这样的明确含义，char或unsigned char均可采用。这些函数并不会把null字符与其他字符区别对待。

C89修正案1增加了5个新函数，用于操纵宽字符数组，它们是由wchar_t *类型的指针所指定的。这些函数是在头文件wchar.h中声明的，它们的名称均以字母wmem开头。宽字符的顺序就是底层整数类型wchar_t中的整数顺序。这些函数不会对宽字符进行解释，因此不会出现编码错误。

参考：wchar_t 第11.1节；宽字符 第2.1.4节。

14.1 memchr、wmemchr

语法概要

```
#include <string.h>
void *memchr( const void *ptr, int val, size_t len );
```

```
#include <wchar.h>
wchar_t *wmemchr( const wchar_t *ptr, wchar_t val, size_t len );
```

memchr函数在ptr的前len个字符中搜索val值的第一次出现。如果找到val，它返回一个指向包含val值的第一个字符的指针，否则返回一个null字符。每个字符c根据表达式(unsigned char) c == (unsigned char) val与val进行比较。读者可以参考strchr（第13.5节）。尽管这个函数所返回的指针被声明为指向非const对象，但是如果它的第一个参数是个const对象，这个指针也可以指向const对象。

wmemchr函数（C89修正案1）在ptr的前len个宽字符中查找val的第一次出现。它返回指向它所找到的第一个宽字符的指针。如果没有找到，它返回一个null指针。

在传统C中，memchr的签名是

```
#include <memory.h>
char *memchr(char *ptr, int val, int len );
```

14.2 memcmp、wmemcmp

语法概要

```
#include <string.h>
int memcmp( const void *ptr1, const void *ptr2, size_t len );
```

（续）

语法概要

```
#include <wchar.h>
int wmemcmp(
  const wchar_t *ptr1, const wchar_t *ptr2, size_t len );
```

memcmp函数把ptr1的前len个字符与ptr2的前len个字符进行比较。如果第一个字符串按词典顺序小于第二个字符串，memcmp就返回一个负值。如果第一个字符串按词典顺序大于第二个字符串，memcmp就返回一个正值。如果两者相等，这个函数返回0。读者可以参考strcmp（第13.2节）。

wmemcmp函数（C89修正案1）对宽字符数组执行与上面相同的操作。宽字符的顺序就是底层整数类型wchar_t的整数顺序。根据ptr1的宽字符序列是小于、等于或大于ptr2的宽字符序列，这个函数的返回值分别是负值、0或正值。

旧式的C编译器可能还采用bcmp函数，它也是对两个字符串进行比较。如果两者相等，这个函数返回0，否则返回一个非零值。它不会比较大于或小于关系。在传统C中，bcmp和memcmp的签名是：

```
#include <memory.h>
int bcmp( char *ptr1, char *ptr2, int len );
int memcmp( char *ptr1, char *ptr2, int len );
```

14.3 memcpy、memccpy、memmove、wmemcpy、wmemmove

语法概要

```
#include <string.h>
void *memcpy (void *dest, const void *src, size_t len);
void *memmove(void *dest, const void *src, size_t len);
```

```
#include <wchar.h>
wchar_t *wmemcpy(
  wchar_t *dest, const wchar_t *src, size_t len);
wchar_t * wmemmove(
  wchar_t *dest, const wchar_t *src, size_t len);
```

memcpy和memmove函数（标准C）都从src复制len个字符到dest，并返回dest的值。两者的区别在于memmove函数在内存区域重叠的情况下也能正确地执行操作。也就是说，memmove的操作方式就像首先把源区域复制到一块独立的临时区域，然后再复制到目标区域（实际上，在实现memmove时并不需要使用临时区域）。如果源内存区域和目标内存区域重叠，memcpy的行为是未定义的，尽管有些版本的memcpy实际上采用了“复制到临时区域”的方式。如果memcpy和memmove均可用，程序员可以认为memcpy的速度更快一些。另外，读者可以参考strcpy（第13.3节）。

wmemcpy和wmemmove（C89修正案1）分别类似于memcpy和memmove，但它们是对宽字符数组进行操作。这两个函数都返回dest的值。

除了memcpy之外，旧式的C编译器还可能使用memccpy和bcopy函数。memccpy函数也是从src复制len字符到dest，但是当它复制了一个值为val的字符之后会立即停止复制。如果全部len个字符都被复制，memccpy返回一个null指针，否则它就返回一个指向dest中被复制的那个val字符之后的位置的指针。bcopy函数的工作方式与memcpy相似，但源操作数和目标操作数正好相反。在传统C中，这些函数的签名是：

```
#include <memory.h>
char *memcpy( char *dest, char *src, int len );
char *memccpy(char *dest, char *src, int val, int len);
char *bcopy( char *src, char *dest, int len );
```

14.4 memset、wmemset

语法概要

```
#include <string.h>
void *memset( void *ptr, int val, size_t len );
```

```
#include <wchar.h>
wchar_t *wmemset( wchar_t *ptr, int val, size_t len );
```

memset函数把val复制到从ptr开始的len个字符中。ptr所指定的字符被认为是unsigned char类型。这个函数返回ptr的值。

wmemset函数（C89修正案1）与memset相似，区别在于它所填充的是宽字符数组。

旧式C编译器还可能使用更具限制性的bzero函数，它把0复制到从ptr开始的len个字符。在传统C中，这些函数的签名是：

```
#include <memory.h>
char *memset( char *ptr, int val, int len );
void bzero( char *ptr,int len );
```



第15章
输入/输出工具

C提供了丰富而实用的I/O工具集，这些工具基于流的概念，流可能是文件，也可能是数据的其他来源或消费者，包括终端或其他物理设备。数据类型FILE（它和其他I/O工具都是在stdio.h中定义的）保存了和流有关的信息。FILE类型的对象是通过调用fopen创建的，本章所描述的大多数I/O工具都接受一个指向这种类型的指针（文件指针）作为参数。

FILE对象所包含的信息包括流内部的当前位置（文件位置）、指向相关联的缓冲区的所有指针以及错误提示和文件尾提示。流一般使用缓冲，除非它们与交互式设备相关联。程序员可以使用setvbuf工具，对缓冲施加一些控制。但是，一般而言，流的实现效率很高，程序员不必担心它的性能。

流有两种基本形式：文本流和二进制流。文本流由划分为行的字符序列所组成，每一行包含0个或多个字符以及一个换行符'\n'。当文本行完全由标准字符集的字符所组成时，文本流是可以移植的。一个特定的C运行时函数库实现的底层硬件和软件成分可能使用不同的表示形式来表示文本文件（尤其是行末提示符），但是运行时函数库必须把这些表示形式映射为标准表示形式。标准C要求编译器支持文本行的长度至少为254个字符（包含结尾的换行符）。

二进制流是char类型的数据值序列。由于任何C数据值都可以映射为char类型的值的数组，因此二进制流可以透明地记录内部数据。编译器也可以选择不区分文本流和二进制流，只要这样做对它而言更为方便。

当C程序开始执行时，有3个预定义的文本流已经被打开：标准输入（stdin）、标准输出（stdout）和标准错误（stderr）。

参考：fopen 第15.2节；setvbuf 第15.3节；标准字符集 第2.1节。

宽字符输入和输出 C89修正案1在C语言中增加了一个宽字符I/O工具。wchar.h所提供的新的宽字符输入/输出函数对应于原来的字节输入/输出函数，不同之处在于底层程序数据类型（和流元素）是宽字符（wchar_t）而不是字符（char）。事实上，这些宽字符I/O函数的实现可能会在宽字符与外部媒介所存储的多字符序列之间来回转换，但它们在程序员面前一般是透明的。

C89修正案1并不是为宽字符I/O创建新的流类型，而是在现有的文本流和二进制流中添加了一个方向（orientation）。打开一个流，并在这个流上执行任何输入/输出操作之前，它是没有方向的。流可以是宽字符定向的或字节定向的，取决于第一个输入输出操作是通过宽字符函数还是字节函数执行的。当一个流被定向之后，就只能使用同种方向的I/O函数，否则结果就是未定义的。fwide函数（第15.2节）可以用于设置或测试流的方向。

当一个文件的外部表示形式是个多字节字符序列时，多字节字符的有些规则在文件中被放宽：

1. 文件的多节字编码可以包含嵌入的null字符。
2. 文件并不需要以初始的转换状态开始或结束。

不同的文件对于宽字符可能使用不同的多字节字符编码。文件的编码在逻辑上是内部转换

状态的一部分，它是在内部转换状态初次绑定时根据区域设置的LC_CTYPE分类而确立的，而不是在第一次调用宽字符输入/输出函数之后才确定的。当一个文件的转换状态（以及编码规则）被绑定之后，LC_CTYPE的设置不再影响相关联的流上所执行的转换。

由于宽字符和多字节字符之间的转换可能存在与之相关联的状态，因此每个宽字符定向的流都与一个隐藏的mbstate_t对象相关联。从概念上说，输入/输出期间的转换是通过使用隐藏转换状态调用mbrtowc或wcrtomb而发生的。fgetpos和fsetpos函数必须通过文件位置记录这个转换状态。宽字符输入/输出期间的转换可能因为编码错误而失败，在这种情况下，errno的值被设置为EILSEQ。如果允许文件使用多种编码，流的编码很可能是mbstate_t对象的一部分，或至少被它所记录。

参考：转换状态 第2.1节；EILSEQ 第11.2节；fgetpos和fsetpos 第15.5节；mbrtowc 第11.7节；mbstate_t 第11.1节；多字节字符 第2.1.5节；定向 第15.2.2节；wcrtomb 第11.7节；宽字符 第2.1.5节。

15.1 FILE、EOF、wchar_t、WEOF

语法概要

```
#include <stdio.h>

typedef ... FILE ...;
#define EOF (-n)
#define NULL ...
#define size_t ...
```

```
#include <wchar.h>

typedef ... wchar_t;
typedef ... wint_t;
#define WEOF ...
#define WCHAR_MAX ...
#define WCHAR_MIN ...
#define NULL ...
#define size_t ...
```

在标准I/O库中大量使用的FILE类型表示流的控制信息。它既可以用于读取字符定向的文件，也可以用于读取宽字符定向的文件。

按照常规，EOF的值表示文件尾，意味着所有的输入数据均已耗尽。在大多数传统的编译器中，这个值是-1，但是，标准C只要求它是个负的整型常量表达式。由于EOF有时也可能表示其他问题，所以当返回EOF时最好使用feof工具（第15.14节）判断是否真的遇到了文件尾。在宽字符I/O中所使用的WEOF宏（C89修正案1）与字节I/O的EOF具有相同的用途。它的值是wint_t类型（并不一定是wchar_t），并且不一定是负值。wchar_t类型可以表示的最大值是WCHAR_MAX，可以表示的最小值是WCHAR_MIN。

为了方便起见，头文件stdio.h和wchar.h定义了size_t类型和空指针常量NULL。在标准C中，它们还在stddef.h中进行了定义，它们在多个头文件中存在并不会有什么害处。

参考：wchar_t 第2.1.5节、第11.1节；wint_t 第2.1.5节、第11.1节。

15.2 fopen、fclose、fflush、freopen、fwide

语法概要

```
#include <stdio.h>

FILE *fopen(
  const char * restrict filename, const char * restrict mode);
int fclose(FILE * restrict stream);
int fflush(FILE * restrict stream);
FILE *freopen(
  const char * restrict filename,
  const char * restrict mode,
  FILE * restrict stream);

#define FOPEN_MAX …
#define FILENAME_MAX …
```

```
#include <wchar.t>

int fwide(FILE * restrict stream, int orient);
```

fopen函数接受文件名和文件模式为参数，这两个参数均是以字符串的形式指定的。编译器按照自己的特有方式使用文件名，打开或创建一个文件，并把它与一个流相关联(FILENAME_MAX宏的值表示文件名的最大长度。如果没有实际的最大值，就表示一个合适的长度)。这个函数返回一个FILE *类型的指针，用于标识这个流，以便在这个流上进行其他输入/输出操作。如果检测到错误，fopen函数在errerno中存储一个错误代码，并返回一个null指针。C并未指定能够同时打开的流的最大数量。在标准C中，这个值是由FOPEN_MAX宏所指定的，这个值至少为8（包括3个预定义的流）。在C89修正案1中，fopen所返回的流并没有定向，因此可以在它上面执行字符定向或宽字符定向的输入/输出操作（但不能同时执行）。

fclose函数以一种合理且有序的方式关闭一个打开的流，包括清空它的所有内部数据缓冲区。如果检测到错误，fclose函数返回EOF，否则就返回0。

例子

下面这些函数打开和关闭常规的文本文件。它们处理错误条件，并根据需要打印诊断信息。它们的返回值与fopen和fclose的返回值匹配：

```
#include <errno.h>
#include <stdio.h>

FILE *open_input(const char *filename)
    /* Open filename for input; return NULL if problem */
{
    FILE *f;
    errno = 0;
    /* Functions below might choke on a NULL filename. */
    if (filename == NULL) filename = "";
    f = fopen(filename,"r");    /* "w" for open_output */
    if (f == NULL)
        fprintf(stderr,
                "open_input(\"%s\") failed: %s\n",
                filename, strerror(errno));
    return f;
}
```

```
int close_file(FILE *f)
    /* Close file f */
{
    int s  = 0;
    if (f == NULL) return 0; /* Ingore this case */
    errno = 0;
    s = fclose(f);
    if (s == EOF) perror("Close failed");
    return s;
}
```

fflush函数清空与输出相关联的所有缓冲区，或者更新流参数。流仍然保持打开。如果检测到任何错误，fflush函数返回EOF，否则就返回0。fflush一般只用于异常情况。在正常情况下，由fclose和exit负责对输入缓冲区进行刷新。

freopen函数接受一个文件名、一种模式和一个打开的流为参数。它首先尝试关闭stream，就像调用fclose函数一样，但是，它将忽略在此过程中所发生的任何错误。然后，它使用filename和mode打开一个新文件，就像调用fopen一样，只是把这个新流与stream相关联，而不是创建一个新的FILE *类型的值。如果调用成功，freopen函数返回stream，否则（如果重新打开文件失败）返回一个null指针。freopen的一个主要用途是把一个标准输入输出流（stdin、stdout和stderr）与另一个文件进行重新关联。在C89修正案1中，freopen可以删除流以前的任何定向。

参考：EOF 第15.1节；exit 第19.5节；stdin第15.4节。

15.2.1 文件模式

表15-1所显示的值可以作为fopen和freopen的文件打开模式。

表15-1 fopen和freopen的模式指定符

模式[a]	含义
"r"	打开一个现有的文件，用于输入
"w"	创建一个新文件或截断一个现有的文件，用于输出
"a"	创建一个新文件或添加到一个现有文件的后面，用于输出
"r+"	打开一个现有的文件，从文件的起始位置开始进行更新（既可读又可写）
"w+"	创建一个新文件或截断一个现在的文件，用于更新
"a+"	创建一个新文件或添加到一个现有文件的后面，用于更新

a 所有的模式后面可以添加字母b，表示当前的流用于保存二进制数据而不是字符数据。

当一个文件打开用于更新时（模式字符串中存在+符号）时，它所产生的流既可用于读取，也可用于写入。但是，在输入操作之后，如果不调用fsetpos、sfeek、rewind或fflush函数，就不能接着进行输出操作。同理，在输出操作之后，如果不调用fsetpos、sfeek、rewind、fflush函数或者执行遇到文件尾的输入操作，就不能接着进行输入操作（这些操作会清空所有的内部缓冲区）。

标准C允许在表15-1所列出的任何类型后面添加字符b，表示创建“二进制”（而不是“文本”）流（两者的区别在UNIX系统中较为模糊，因为它们是按相同的方式处理的。其他操作系统就没有这么好的运气了）。标准C还允许任何“更新”文件类型采用二进制模式。在流的模式指定符中，b可以出现在 + 的之前或之后。

在标准C中，除了前面所列出的模式之外，模式字符串还可以包含其他字符。编译器可以使用这些附加字符指定流的其他属性。例如：

```
f = fopen("C:\\work\\dict.txt","r,access=lock");
```

表15-2列出了每种流模式的一些属性。

表15-2 fopen模式的属性

属性	模式					
	r	w	a	r+	w+	a+
指定文件必须已存在	是	否	否	是	否	否
现有文件的内容将丢失	否	是	否	否	是	否
从允许的流进行读取	是	否	否	是	是	是
写入到允许的流	否	是	是	是	是	是
从流的末尾开始写入	否	否	是	否	否	是

15.2.2 文件的定向

fwide函数（C89修正案1）用于测试和（或）设置流的定向。在调用之后，这个函数根据stream是宽字符定向的、字节定向的还是未定向，分别返回正值、负值或零。orient参数决定了fwide函数是否首先尝试设置定向。如果orient为0，就不会尝试设置定向，这个函数的返回值反映了在调用时这个流的定向。如果orient为正值，则fwide试图设置宽字符定向。如果orient为负值，则fwide函数试图设置字节定向。只有当提供给这个函数的流并没有设置定向时，这些尝试才可能是成功的。也就是说，这个流是由fopen或freopen打开的。否则，这个函数就会不加修改地返回这个流的定向状态。

例子

在使用宽字符定向的流时，在fopen被调用时使用fwide建立流的定向是一种好的思路。下面这个函数用一种指定的模式打开一个特定的文件，并把它设置为一种特定区域的宽字符定向。如果成功，这个函数返回一个文件指针，否则返回NULL。

```
FILE *fopen_wide(
    const char *filename, /* file to open */
    const char *mode,     /* mode for open */
    const char *locale)/* locale for encoding */
{
    FILE *f = fopen(filename, mode);
    if (f != NULL) {
        char *old_locale = setlocale(LC_CTYPE, locale);
        if (old_locale == NULL || fwide(f, 1) <= 0) {
            fclose(f);   /* setlocale or fwide failed */
            f = NULL;
        }
        /* return locale to its original value */
        setlocale(LC_CTYPE, old_locale);
    }
    return f;
}
```

如果使用了多字节编码（如果有），就会在建立流的定向时确定这种编码。它将受到流的定向建立时当前区域的LC_TYPE分类设置的影响。

15.3 setbuf、setvbuf

语法概要

```
#include <stdio.h>
int setvbuf(
  FILE * restrict stream,
  char *b restrict uf,
  int bufmode,
  size_t size );
void setbuf(
  FILE * restrict stream,
  char * restrict buf );
#define BUFSIZ ...
#define _IOFBF ...
#define _IOLBF ...
#define _IONBF ...
```

这些函数允许程序员在默认的缓冲区无法满足要求的罕见情况下控制流的缓冲区策略。这些函数必须在流被打开之后并且在任何数据被读取或写入之前被调用。

setvbuf函数是从UNIX System V所采纳的一个更通用的函数。第1个参数是被控制的流；第2个参数（如果不为空）是个字符数组，用于代替自动产生的缓冲区；bufmode参数指定了缓冲区的类型；size参数指定了缓冲区的长度。如果调用成功，这个函数返回0。如果参数不正确或者请求无法满足，这个函数返回非零值。

_IOFBF、_IOLBF和_IONBF宏被展开为bufmode可以使用的值。如果bufmode为_IOFBF，这个流被完全缓冲。如果bufmode为_IOLBF，当换行符被写入或者缓冲区已满时，缓冲区才被刷新。如果bufmode为_IONBF，则这个流不会被缓冲。如果请求了缓冲并且buf并不是null指针，则buf所指定的数组的长度应该是size个字节，并且将代替自动产生的缓冲区。BUFSIZ常量是缓冲区长度的“适当”值。

setbuf函数是setvbuf函数的简化版本。下面这个表达式

```
setbuf(stream,buf)
```

相当于下面这个表达式

```
((buf==NULL) ?
  (void) setvbuf(stream,NULL,_IONBF,0) :
  (void) setvbuf(stream,buf,_IOFBF,BUFSIZ))
```

参考：EOF 第15.1节；fopen 第15.2节；size_t 第11.1节。

15.4 stdin、stdout、stderr

语法概要

```
#include <stdio.h>
#define stderr ...
```

（续）

语法概要

```
#define stdin  ...
#define stdout ...
```

表达式stdin、stdout和stderr的类型是FILE *。在应用程序开始之前，它们的值就已经确定为一些标准的文本流。stdin指向一个输入流，它是程序的“正常输入”。stdout指向一个输出流，它是程序的“正常输出”。stderr指向一个输出流，它表示错误信息以及程序所遇到的其他意料之外的输出。在交互式环境中，这3个流一般都与用于启动程序的终端相关联，并且除了stderr之外，都是使用了缓冲。

这些表达式通常并不是左值，在任何情况下它们都不应该通过赋值被修改。freopen函数（第15.2节）可以用于更改这些流。

例子

表达式stdin、stdout和stderr常被定义为静态或全局流描述符的地址：

```
extern FILE __iob[FOPEN_MAX];
...
#define stdin  (& __iob[0])
#define stdout (&__iob[1])
#define stderr (&__iob[2])
```

UNIX系统特别提供了一些方便的方法，当应用程序启动时，把这些标准流与文件或其他程序相关联。这样，当它们根据一些标准的约定使用时，可以实现更为强大的功能。

在C89修正案1中，stdin、stdout和stderr在C程序启动时没有定向。因此，可以调用fwide函数（第15.2节）或者在这些流上使用宽字符输入/输出函数，使这些流用于宽字符的输入/输出。

15.5 fseek、ftell、rewind、fgetpos、fsetpos

语法概要

```
#include <stdio.h>

int fseek(
  FILE * restrict stream, long int offset, int wherefrom);
long int ftell(FILE * restrict stream);
void rewind(FILE * restrict stream);
#define SEEK_SET 0
#define SEEK_CUR 1
#define SEEK_END 2

typedef ... fpos_t ...;
int fgetpos( FILE * restrict stream, fpos_t *pos );
int fsetpos( FILE * restrict stream, const fpos_t *pos );
```

本节的函数允许随机访问文本和二进制流。一般情况下，这些流与文件相关联。

15.5.1 fseek和ftell

ftell函数接受一个用于输入或输出的流，并返回这个流中的一个位置，这个值的形式适合作为fseek的第2个参数。如果在调用fseek时使用ftell所保存的结果，将使流的位置重置为调用ftell

时的当前文件位置。

对于二进制文件，ftell的返回值将是当前文件位置之前的字符数量。对于文本文件，这个返回值因编译器而异。返回值必须可以在fseek中使用，并且0L必须是文件起始位置的一种表示形式（但并不一定是唯一的一种）。

如果ftell遇到了错误，它返回-1L，并把errno设置为一个由编译器所定义的正值。由于-1L可能被看成是个合法的文件位置，因此必须检查errno来验证这个错误。可能导致ftell失败的条件包括试图在一个连接到终端的流中查找文件位置，或者试图报告一个无法用long int类型的对象表示的位置。

fseek函数允许在（打开的）stream中进行随机访问。第2个和第3个参数指定了一个文件位置：offset是个有符号（long）整数，指定了字符的数量（对于二进制流而言）；wherefrom是个“定位代码”，表示应该从文件中的哪个位置开始计算offset。流的定位如下表所示。如果调用成功，fseek返回0，如果遇到了错误，它就返回一个非零值（errno的值不会改变）。任何文件尾提示标志均被清除，任何ungetc的效果均被还原。标准C定义了常量SEEK_SET、SEEK_CUR和SEEK_END，表示wherefrom的值。使用非标准编译器的程序员必须使用指定的整数或者定义这些宏。

对一个二进制文件进行重新定位时，新位置是由下面的表格给出的：

如果wherefrom是	则新位置是
SEEK_SET或0	从文件起始位置开始的offset个字符
SEEK_CUR或1	从文件的当前位置开始的offset个字符
SEEK_END或2	从文件尾开始的offset个字符（负值指定了文件尾之前，正值指定了对文件进行扩展，使它包含未指定的内容。）

对于二进制流，标准C并不要求编译器支持wherefrom参数所使用的SEEK_END常量的“有意义”的值。下面是标准C所允许的对文本流所进行的更有限的调用集合：

调用形式	位置（文本）流
fseek(stream, 0L, SEEK_SET)	在文件的起始位置
fseek(stream, 0L, SEEK_CUR)	和原来相同的位置（即调用没有效果）
fseek(stream, 0L, SEEK_END)	在文件的尾部
fseek(stream, *ftell - pos*, SEEK_SET)	前一次在stream上调用ftell时所返回的位置

这些限制是认识到文本文件中的位置可能无法直接映射到文件的内部表示形式。例如，一个位置可能要求一个记录号和该记录内的一个偏移值（但是，标准C要求编译器支持对文本文件调用fseek(stream, 0L, SEEK_END)，但对于二进制文件则不要求提供“有意义”的支持）。

在C89修正案1中，在宽字符定向的流上所执行的文件定位操作必须满足二进制或文本文件流的所有限制。fseek和ftell函数一般并没有强大到足以支持宽字符定向的流，即使是查找流的头部和尾部这样的简单操作。程序员应该使用下一节所描述的fgetpos和fsetpos函数对宽字符定向的流执行操作。

rewind函数把流重置为起始位置。根据标准C的定义，调用rewind(stream)相当于

```
(void) fseek(stream, 0L, SEEK_SET)
```

15.5.2 fgetpos和fsetpos

fgetpos和fsetpos函数是标准C新增的。增加它们的目的是为了处理那些因为过于庞大而无法由long int类型的整数来定位的文件（ftell和fseek使用long int类型来定位）。

fgetpos函数把当前文件位置存储在pos所指向的对象中。如果调用成功，这个函数返回0。如果遇到了错误，它返回一个非零值，并在errno中存储一个因编译器而异的正值。

fsetpos函数根据*pos中的值设置当前文件位置，后者必须是前面在同一个流上调用fgetpos所返回的一个值。fsetpos将会还原ungetc或ungetwc的任何效果。如果调用成功，这个函数返回0。如果遇到了错误，它返回一个非零值，并在errno中存储一个因编译器而异的正值。

在C89修正案1中，fgetpos和fsetpos所使用的文件位置对象必须包含与宽字符定向的流相关联的隐藏转换状态的表示形式（即一个mbstate_t类型的值）。在重新定位操作之后，需要使用这个状态以及文件中的位置来解释接下来的多字节字符。

在宽字符定向的输出流中，使用fsetpos设置输出位置然后写入1个或多个多字节字符将导致文件中后面的所有多字节字符都变为未定义。这是因为输出操作可能会重写一个现有的多字节字符的一部分，或者可能会改变转换状态，导致后面的多字节字符无法被正确地解释。

参考：mbstate_t 第11.1节；ungetc 第15.6节。

15.6 fgetc、fgetwc、getc、getwc、getchar、getwchar、ungetc、ungetwc

语法概要

```
#include <stdio.h>

int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar(void);
int ungetc(int c, FILE *stream);
```

```
#include <stdio.h>
#include <wchar.h>

wint_t fgetwc(FILE *stream);
wint_t getwc(FILE *stream);
wint_t getwchar(void);
wint_t ungetwc(wint_t c, FILE *stream);
```

fgetc函数接受一个输入流作为参数。它从这个流中读取下一个字符，并把它作为一个int类型的值返回。内部的流位置指示符的值将会增加。连续调用fgetc将会返回输入流中的连续字符。如果发生了错误或者流已经位于文件的尾部，fgetc返回EOF。在这种情况下，程序员应该使用feof和（或）ferror工具来判断是否真正到达了文件尾。

getc的功能与fgetc相同，但为了提高效率，它一般被实现为宏。stream参数不应该具有任何副作用，因为它可能会被多次使用。

getchar相当于getc(stdin)。和getc一样，getchar也常常被实现为宏。

在C89修正案1中，fgetwc、getwc和getwchar都与对应的字节定向的版本相似（包含它们可能采用宏的实现），但它们从输入流中读取和返回下一个宽字符。如果遇到错误或者到达了文

件尾，它们返回WEOF。如果遇到编码错误，就会在errno中存储EILSEQ这个值。读取宽字符涉及从多字节字符转换为宽字符。这个过程就像使用流的内部转换状态调用mbrtowc所执行的操作一样。

ungetc函数使字符c（转换为unsigned char类型）被压回到指定的输入流，使它将被下一次在这个流上所调用的fgetc、getc或getchar所返回。如果压入了多个字符，它们将会以与压入顺序相反的顺序返回（即最后一个被压入的字符首先被返回）。当成功压回字符时，ungetc返回c。如果失败，它就返回EOF。在流中成功执行了文件定位操作（fseek、fsetpos或rewind）之后将会丢弃所有被压回的字符。在读取（或丢弃）所有被压回的字符之后，文件位置就与压回字符之前的位置相同。

只要流进行了缓冲并且从这个流上前一次执行fseek、fopen或freopen之后至少读取了1个字符，就可以保证能够压回1个字符。试图把EOF这个值作为字符压回到流中不会对流产生任何影响，并且会返回EOF。调用fsetpos、rewind、fseek或freopen将会在流中删除所有被压回的字符，但并不会影响与这个流相关联的任何外部存储。

ungetc函数可以用于实现像scanf这样的输入扫描操作。程序可以“预先查看”下一个输入字符，也就是先读取它，如果不合适，再把它压回到流中（但是，scanf以及其他库函数不允许抢占程序员对ungetc的使用。也就是说，即使在调用scanf或类似的函数之后，程序员至少可以保证仍然有1个被压回的字符）。

ungetwc函数（C89修正案1）与ungetc类似。

参考：EOF 第15.1节；feof 第15.14节；fseek 第15.5节；fopen 第15.2节；freopen 第15.2节；scanf 第15.8节；stdin 第15.4节。

15.7 fgets、fgetws、gets

语法概要

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
char *gets(char *s);
```

```
#include <stdio.h>
#include <wchar.h>
wchar_t *fgetws(wchar_t *s, int n, FILE *stream);
```

fgets函数接受3个参数：指向一个字符数组起始位置的指针s、计数值n和一个输入流。字符从输入流读取到s，直到遇见换行符或文件尾，或者在未遇到换行符或文件尾的情况下读取了n－1个字符。然后，在这个字符数组中已读取字符的后面会添加一个表示结尾的null字符。如果输入由于遇到换行符而终止，则这个换行符也存储在数组中，就在null字符之前。在成功完成之后，这个函数返回参数s。

如果在输入流中读取任何字符之前就遇到了文件尾标志，fgets返回一个null指针，数组s的内容将不会变化。如果在输入操作过程中遇到了错误，则fgets返回一个null指针，并且数组s的内容是不确定的。如果这个函数返回NULL，应该使用feof工具（第15.14节）来判断是否真正遇到了文件尾。

gets函数从标准输入流stdin中读取字符到字符数组s中。但是，和fgets不同，当输入是因为遇见换行符而终止时，gets会丢弃这个换行符，并不会把它存储在s中。使用gets存在一定的危险，因为它很可能导致输入长度超过字符数组的长度。fgets函数更为安全，因为它不会在s中存储超过n个的字符。

fgetws函数（C89修正案1）与fgets相似，但它是对宽字符定向的输入流进行操作，把宽字符存储在s，包括在尾部添加一个null宽字符。gets不存在对应的宽字符版本，这也在另一方面暗示应该避免使用gets。

参考：feof 第15.14节；stdin 第15.4节。

15.8 fscanf、fwscanf、scanf、wscanf、sscanf、swscanf

语法概要

```
#include <stdio.h>

int fscanf(
  FILE * restrict stream, const char * restrict format, ...);
int scanf(
  const char * restrict format, ...);
int sscanf(
  char *s, const char * restrict format, ...);
```

```
#include <stdio.h>
#include <wchar.h>

int fwscanf(
  FILE * restrict stream,
  const wchar_t * restrict format, ...);
int wscanf(
  const wchar_t *format, ...);
int swscanf(
  wchar_t *s, const wchar_t *format, ...);
```

fscanf函数对格式化的输入文本进行解析，从第1个参数所指定的流中读取字符，并根据格式控制字符串对字符序列进行转换。根据控制字符中的内容，可能还需要额外的参数。控制字符串之后的每个参数都必须是指针。从输入流所读取的经过转换的值被存储到这些指针所指定的对象中。

scanf和sscanf与fscanf相似。对于scanf，字符是从标准输入流stdin读取的。对于sscanf，字符是从字符串s读取的。当sscanf读取到字符串s的尾部之后时，它就按照fscanf和scanf在遇到文件尾时一样的方式进行操作。

输入操作可能由于输入流到达了文件尾或者由于控制字符串和一个从输入流所读取的字符发生冲突而永久终止。这些函数所返回的值是操作结束（因为任何原因）之前成功执行的赋值的数量。如果未发生冲突并且未进行任何赋值之前输入流就到达了文件尾，则这些函数返回EOF。当冲突发生时，导致冲突发生的字符将保留在未读取状态，将由下一次输入操作所处理。

C89修正案1定义了一组宽字符格式化输入函数，与fscanf、scanf和sscanf对应。这些新的wscanf函数群使用了宽字符控制字符串，并期望从输入流中所读取的是宽字符序列。任何来自外部文件的底层多字节序列的转换对程序员是透明的。在下面的描述中，重点将讨论字节定向

的函数。除非另有说明，不然宽字符定向的函数的行为可以通过用“宽字符”代替“字符”或“字节”来推断。

修正案1还对标准C的格式化字符串进行了扩展，允许向s、c增加长度指定符l，并增加了[转换操作，表示相关联的实参是个指向宽字符或字符的指针。更多的信息详见对这些转换操作的描述。

15.8.1 控制字符串

控制字符串是输入数据的预期格式的蓝图。在标准C中，它是scanf函数群所使用的以起始转换状态开始和结束的多字节字符序列，以及wscanf函数群所使用的宽字符序列。人们可以把这些函数看成是在控制字符串和输入流之间所执行的简单的匹配操作。控制字符串的内容可以分为3个种类：

1. 空白字符：控制字符串中的空白字符使空白字符被读取并丢弃。第1个遇到的非空白输入字符被保留为输入流中下一个将要被读取的字符。注意，如果控制字符串中出现了几个连续的空白字符，其效果相当于只出现1个空白字符。因此，控制字符串中任意的连续空白字符序列将与输入流中任意的连续空白字符序列相匹配，它们的长度可以不同。

2. 转换规范：转换规范以百分号%开始，下面详细描述转换规范语法的剩余部分。从输入流读取的字符数量取决于转换操作。根据经验，转换操作将会持续处理字符，直到：（a）遇到文件尾；（b）遇到空白字符或其他不合适的字符；（c）转换操作所读取的字符数量等于指定的最大字段宽度。被处理的字符一般被转换（例如转换为数值）并存储在控制字符串之后的指针参数所指定的位置。

3. 其他字符：除空白字符或百分号之外的任何字符都必须与输入流中的下一个字符匹配。如果不匹配，就会出现冲突，转换操作将会终止，导致冲突发生的那个输入字符将保留在输入流中，供这个流的下一次输入操作所读取。

指针参数的数量应该正确，并且每个参数都应该具有正确的类型，以满足控制字符串所指定的转换规范。如果出现了过多的参数，多余的参数将被忽略。如果参数过少，其结果是未定义的。如果转换规范出现了任何形式错误，其行为也是未定义的。在每个转换规范所执行的操作之后会出现序列点。

15.8.2 转换规范

转换规范以百分号%开头。在百分号之后，可以按顺序出现下面这些转换规范元素：

1. 一个可选的取消赋值标志，写成一个星号（*）。如果它出现在一个在正常情况下将执行赋值的转换操作中，则从输入流读取和处理的字符是按该操作的通常方式进行的，但并不会执行赋值，也不会消耗任何指针参数。

2. 一个可选的最大字段宽度，用一个正的十进制整数表示。

3. 一个可选的长度规范，用hh、h、l（字母）、ll（均为字母）、j、z、t或L这些字符序列之一来表示。表15-3列出了它们可以应用的转换操作。长度规范hh、ll、j、z和t是C99新增的。

4. 必需的转换操作（或转换指定符），用单个字符表示（仅有1个例外）：a、c、d、e、f、g、i、n、o、p、s、u、x、%或[。例外的是[操作，它导致到下一个]为止的所有字符成为转换规范

的一部分。

fscanf的转换规范在语法和含义上与fprintf的转换规范相似，但存在一些差别。最好把fprintf和fscanf的控制字符串语法看成是粗略相似，不能使用fprintf的转换规范文档来指导fscanf转换规范的用法（或反之）。

例子

下面是fscanf和fprintf的转换所存在的一些不同之处：

[转换操作是fscanf所特有的。

fscanf并不承认fprintf所接受的任何类型的精度规范，也不承认fprintf所接受的下面这些标志字符-、+、空格、0和#的任何一个。

显式指定的字段宽度对于fprintf而言是最小值，但对于fscanf而言是最大值。

fprintf允许使用经过计算的参数指定字段宽度，也就是用星号来指定字段宽度。但是，fscanf把星号用于其他目的，即取消赋值。这一点也许是它们最显著的区别所在。

除此之外，所有的转换操作会在转换之前跳过所有的起始空白字符。这些起始空白字符并不被算在最大字段宽度中。在正常情况下，这些转换操作都不会跳过缀尾的空白字符。缀尾的空白字符（例如终止某一行输入的换行符）将保留为未读取状态，除非它确实与控制字符串匹配（这种做法较为怪异，因为控制字符串中的空白字符将试图匹配输入中的许多空白字符，从而导致读取到换行符之后）。

程序员无法直接判断控制字符串中的字面值字符的匹配是否成功，也无法直接判断涉及消除赋值的转换操作是成功还是失败。这些函数的返回值只反映了成功执行的赋值的数量。

转换操作非常复杂。表15-3提供了一个简单的总结，并在后面进行了详细的讨论。

表15-3 输入转换（scanf、fscanf、sscanf）

转换字母	长度指定符	参数类型	输入格式	
d	无	int *	[-	+]dd...d
i[a]	hh	char *	[-	+][0[x]]dd...d[b]
	h	short *		
	l	long *		
	ll[c]	long long *		
	j	intmax_t *		
	z	size_t *		
	t	ptrdiff_t *		
u	无	unsigned *	[-	+]dd...d
o	hh	unsigned char *	[-	+]dd...d[c]
x	h	unsigned short *	[-	+][0x]dd...d[d]
	l	unsigned long *		
	ll[c]	unsigned long long *		
	j	uintmax_t *		
	z	size_t *		
	t	ptrdiff_t *		
c	无	char *	固定宽度的字符序列。如果使用l，必须是多字节	
	l[a]	wchar_t *		
s	无	char *	非空白字符序列。如果使用l，必须是多字节	

（续）

转换字母	长度指定符	参数类型	输入格式
	l[a]	wchar_t *	
p[a]	无	void **	字符序列，就像在fprintf中用%p所输出的字符
n[a]	无	int *	无。被读取的字符数量存储在参数中
	hh	char *	
	h	short *	
	l	long *	
	ll[e]	long long *	
	j	intmax_t *	
	z	size_t *	
	t	ptrdiff_t *	
a[e], f, e, g	无	float *	任何浮点常量或十进制整数常量，可以有－或+前缀
	l	double *	
	L[a]	long double *	
[	无	char *	来自一个扫描集合的字符序列。如果使用l，必须是多字节
	l[a]	wchar_t *	

a C89所增加。

b 和C常量相同，数值的基数是由第一个数字决定的。

c 这个数值假定是八进制数。

d 这个数值假定是十六进制数，不管是否存在0x。

e C99所增加。

d转换 这个转换执行有符号十进制转换。它消耗1个参数。根据长度指定符，这个参数的类型应该是int *、short *或long *。

它所读取的数值的格式与strtol函数（对于wscanf，则是wcstol）使用10为base参数时所预期的输入格式相同，也就是可选的－或+后面的十进制数字序列。如果输入所表示的值太大，无法用适当长度的有符号整数表示，则它的行为是未定义的。

i转换 这个转换执行有符号整数转换。它消耗1个参数。根据长度指定符，这个参数的类型应该是int *、short *或long *。

它所读取的数值的格式与strtol函数（对于wscanf，则是wcstol）使用0为base参数时所预期的输入格式相同，也就是C整数常量，没有后缀，有可选的－或+符号，以及0（八进制）或0x（十六进制）前缀。如果输入所表示的值太大，无法用适当长度的有符号整数表示，则它的行为是未定义的。

u转换 这个转换执行无符号整数转换。它消耗1个参数。根据长度指定符，这个参数的类型应该是unsigned *、unsigned short *或unsigned long *。

它所读取的数值的格式与strtoul函数（对于wscanf，则是wcstol）使用10为base参数时所预期的输入格式相同，也就是可选的－或+后面的十进制数字序列。如果输入所表示的值太大，无法用适当长度的无符号整数表示，则它的行为是未定义的。

o转换 这个转换执行无符号八进制转换。它消耗1个参数。根据长度指定符，这个参数的类型应该是unsigned *、unsigned short *或unsigned long *。

它所读取的数值的格式与strtoul函数（对于wscanf，则是wcstol）使用8为base参数时所预期

的输入格式相同。也就是可选的－或+后面的八进制数字序列。如果输入所表示的值太大，无法用适当长度的无符号整数表示，则它的行为是未定义的。

x转换 这个转换执行无符号十六进制转换。它消耗1个参数。根据长度指定符，这个参数的类型应该是unsigned *、unsigned short *或unsigned long *。

它所读取的数值的格式与strtoul函数（对于wscanf，则是wcstol）使用16为base参数时所预期的输入格式相同。也就是可选的－或+后面的十六进制数字序列。这个操作接受0123456789abcdefABCDEF中的所有字符作为合法的十六进制数字。如果输入所表示的值太大，无法用适当长度的无符号整数表示，则它的行为是未定义的。

有些非标准的C编译器接受字母X作为与它等价的转换操作。

c转换 这个转换读取1个或多个字符。它消耗1个指针参数，类型必须是char *。如果使用了长度指定符l，则参数的类型是wchar_t *。c转换操作并不会跳过起始的空白字符。应用于输入字符的转换取决于是否存在长度指定符l以及所使用的是scanf还是wscanf。表15-4列出了所有的可能性。

表15-4 c指定符的输入转换

函数	长度指定符	参数类型	输入	转换
scanf	无	char *	字符	无；字符被复制
	l	wchar_t *	多字节字符	转换为宽字符，就像1次或多次调用了mbrtowc函数
wscanf	无	char *	宽字符	转换为多字节字符，就像1次或多次调用了wcrtomb函数
	l	wchar_t *	宽字符	无。宽字符被复制

如果未指定字段宽度，则正好读取1个字符，除非输入流位于文件的尾部。在这种情况下，转换操作将会失败。字符值将被赋值为下一个指针参数所指定的位置。

如果指定了字段宽度，则这个指针参数被假定为指向一个字符数组的起始位置，字段宽度指定了需要读取的字符数量。如果在读取这些数量的字符之前遇到了文件尾，转换操作就将失败。被读取的字符存储在数组的连续位置中。在读取的字符后面不会添加额外的结尾null字符。

s转换 这个转换读取一个字符串。它消耗一个指针参数，这个参数的类型必须是char *，如果使用了长度指定符l（C89修正案1），则参数的类型是wchar_t *。s转换操作总是会跳过起始的空白字符。

这个转换将不断读取字符，直到遇见文件尾，或者直到输入中看到一个空白字符（在这种情况下，这个字符将保留为未读取），或者（如果指定了字段宽度）已经读取了最大指定数量的字符。如果在遇到任何非空白字符之前就遇到了文件尾，则转换操作被认为是失败的。根据是否使用了长度指定符l以及所使用的是scanf还是wscanf，可能会对输入字符执行转换（参见表15-5）。如果在scanf函数中使用了长度指定符l，输入就被第一个空白字符所终止，这是在输入字符被解释为多字节字符之前发生的。

在函数所存储的字符后面总是会添加一个表示结尾的null字符。如果未指定最大字段宽度，s转换操作可能存在危险，因为输入的长度很可能超出字符数组的可用存储空间。

显式指定了字段宽度的s操作与显式指定了字段宽度的c操作不同。c操作并不会跳过空白字符，它将尽可能多地读取指定的字符（或宽字符），直到遇见文件尾。s操作将跳过起始的空白

字符，当它读取了一些非空白的字符（或宽字符）之后再遇见空白字符时，它就将终止，并在它所存储的字符后面添加一个null字符。

表15-5 s指定符的输入转换

函数	长度指定符	参数类型	输入	转换
scanf	无	char *	字符	无。字符被复制
	l	wchar_t *	多字节字符	转换为宽字符，就像调用了mbrtowc一样
wscanf	无	char *	宽字符	转换为多字节字符，就像调用了wcrtomb一样
	l	wchar_t *	宽字符	无。宽字符被复制

p转换 这个转换执行指针转换。它消耗1个参数，参数的类型应该是void **。它所读取的指针值的格式是由编译器所指定的，但通常与printf函数群的%p转换所产生的格式相同。指针的解释也是由编译器定义的，但是如果在程序执行过程中先写出一个指针以后再把它读回，则读回的指针应该与写出时的指针相同。p转换是标准C新增的。

n转换 它并不执行任何转换，也不读取任何字符。它只是把到目前为止调用scanf群的函数所处理的字符的数量写入到参数中。取决于长度指定符，参数的类型必须是int *、short *或long *。n转换是标准C新增的。

a、f、e和g转换 它执行有符号十进制浮点转换。在C99中，允许使用a转换，它与用于输入的f、e和g相同。它消耗1个指针参数。取决于长度指定符，指针参数的类型必须是float *、double *或long double *。

它所读取的数值的格式与strtod函数（如果是wscanf，则是wcstod）的预期输入相同，也就是可选的+或－号后面的十进制或十六进制数字序列，并可以包含小数点和有符号的指数部分（没有小数点的整数也是可以接受的）。输入字符串INF、INFINITY、NAN和NAN（...）（忽略大小写）表示特殊的浮点数。允许接受十六进制浮点数输入的功能是C99新增的。

它所读取的字符被解释为浮点数表示形式，并被转换为一个指定长度的浮点数。如果未读取任何数字，或者至少在指数部分之前没有看到数字，则这个值为0。如果在引入指数部分的字母后面没有看到数字，则这个浮点数表示形式的指数部分就是0。如果输入所表示的值太大或太小，无法用适当长度的浮点数来表示，则返回HUGE_VAL（并具有正确的符号），并把errno设置为ERANGE（在不遵循标准C的编译器中，返回值和errno所设置的值是不可预测的）。如果输入所表示的值并不太小或太小，但没有办法用适当长度的浮点数来表示，就会执行某种形式的四舍五入或截断。

a、f、e和g转换操作是完全相同的，它们都可以被任何风格的浮点表示形式所接受。有些编译器可能接受G和E作为浮点转换字母。

%转换 它期待输入中出现一个百分号。由于百分号用于表示一个转换规范的开始，因此如果需要与一个百分号匹配，就需要连续使用两个百分号。它并不消耗任何指针参数。消除赋值标志、字段宽度以及长度指定符都与%转换操作无关。

[转换 它读取一个字符串，并消耗一个char *或wchar_t *类型（如果使用了长度指定符l）的指针参数。[转换操作并不会跳过起始的空白字符。转换规范准确地指定了哪些字符将被读取作为输入字段的一部分。在控制字符串中，[后面必须是多个字符，并以]终止。到]为止的所

有字符都是这个转换规范的一部分，称为*扫描集*（scanset）。如果 [后面的第1个字符就是 ^ 符号，它具有特殊的含义，表示求反标志，扫描集由未出现在 ^ 和] 之间的所有字符组成。从数学的意义上说，扫描集中的字符被看成是一个集合。

起始的 [和终止的] 之间的任何 [都被看成是其他字符。类似，并未紧随在 [之后的 ^也被看成是其他字符。在标准C中，如果] 紧随 [之后，则它将位于扫描集中，下一个] 将终止这个转换规范。如果] 紧随表示求反标志^的之后，并这个] 并不位于扫描集中，下一个] 将终止这个扫描集。旧式的编译器可能并不支持对出现在转换规范的起始处的] 的特殊处理。

例子

如果转换是……	则扫描集是……
%[abca]	三个字符a、b和c
%[^abca]	除a、c和c之外的所有字符
%[[]	单个 [符号
%[]]	单个] 符号
%[,\t]	空格、逗号和水平制表符

这个转换将连续读取字符，直到遇到文件尾、遇到了不位于扫描集中的字符（此时该字符保留为未读取状态）或者（如果指定了字段宽度）已经读取了最大数量的字符。然后，如果赋值没有被*所取消，输入字符就像s转换一样，存储在参数指针所指定的对象中，并会执行与多字节字符的任何转换（参见表15-5）。然后，被存储的字符后面将添加一个结尾的null字符。长度指定符与 [转换操作无关。

和s转换一样，如果未指定最大字段宽度，[转换操作可能会产生危险，因为输入长度有可能超出字符数组的存储容量。

参考：EOF 第15.1节；fprintf 第15.11节；stdin 第15.4节。

15.9 fputc、fputwc、putc、putwc、putchar、putwchar

语法概要

```
#include <stdio.h>

int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);
```

```
#include <stdio.h>
#include <wchar.h>

wint_t fputwc(wchar_t c, FILE *stream);
wint_t putwc(wchar_t c, FILE *stream);
wint_t putwchar(wchar_t c);
```

fputc函数接受一个字符值和一个输出流为参数。它把字符写入到流的当前位置，并把这个值按int类型返回。接下来对fputc的调用将把特定的字符依次写入到输出流中。如果发生了错误，fputc返回EOF，而不是那个需要写入的字符。

putc函数的操作方式与fputc相似，但它通常被实现为宏。它的参数表达式必须不能有副作用，因为它们可能不止求值1次。

putchar函数向标准输出流stdout写入一个字符。和putc一样，putchar通常也被实现为宏，因此效率极高。调用putchar(c)相当于调用putc(c, stdout)。

C89修正案1增加了宽字符函数fputwc、putwc和putwchar，它们分别对应于字节定向函数。如果发生了错误，它们返回WEOF这个值。如果出现了编码错误，它们还把errno的值设置为EILSEQ。

参考：EOF 第15.1节；stdout 第15.4节。

15.10 fputs、fputws、puts

语法概要

```
#include <stdio.h>

int fputs(const char *s, FILE *stream);
int puts(const char *s);
```

```
#include <stdio.h>
#include <wchar.h>

int fputws(const wchar_t *s, FILE *stream);
```

fputs函数接受一个以null结尾的字符串和一个输出流为参数。它把这个字符串的所有字符写入到输出流中，不包括结尾的null字符。如果发生了错误，fputs返回EOF，否则就返回其他的非负值。

puts函数与fputs相似，区别在于字符总是被写入到标准流stdout。在s中的字符被写入到流中之后，还会写入一个额外的换行符（不管s是否包含了换行符）。

如果s是空字符串，puts函数的一些非标准的UNIX实现将会产生错误，使返回值为不确定。程序员需要注意这种边界情况。

C89修正案1增加了fputws函数，它与fputs相似。这个函数在遇到错误时返回EOF（而不是WEOF）。如果是编码错误，它还把errno设置为EILSEQ。

参考：EOF 第15.1节；stdout 第15.4节。

15.11 fprintf、printf、sprintf、snprintf、fwprintf、wprintf、swprintf

语法概要

```
#include <stdio.h>

int fprintf(
  FILE * restrict stream, const char * restrict format, ... );
int printf(
  const char * restrict format, ...);
int sprintf(
  char * restrict s,
  const char * restrict format, ...);
int snprintf(
  char * restrict s, size_t n,
  const char * restrict format, ...);     // C99
```

（续）

语法概要

```
#include <stdio.h>
#include <wchar.h>

int fwprintf(
  FILE * restrict stream,
  const wchar_t * restrict format, ... );
int wprintf(
  const wchar_t * restrict format, ...);
int swprintf(
  wchar_t *s, size_t n, const wchar_t *format, ...);
```

fprintf函数执行格式化输出，把输出发送给第1个参数所指定的流。第2个参数是格式控制字符串。根据控制字符串的内容，可能还需要其他参数。这个函数根据控制字符串产生一系列的输出字符，这些字符被发送到指定的流。

printf函数与fprintf相关，但它把字符发送给标准输出流stdout。

sprintf函数使输出字符被存储到字符串缓冲区s。在控制字符串所指定的所有字符都被输出之后，它会向s输出一个结尾的null字符。保证sprintf的目标字符串区域足以容纳格式操作所产生的所有字符是程序员的责任。但是，和sprintf不同，swprintf还包含了一个参数，指定了可以写入到输出字符串s的宽字符的最大数量（包含结尾的null字符）。C99增加了snprintf函数，提供了对非宽字符的计数。

如果在输出操作过程中遇到了错误，这些函数返回EOF。否则，它们就返回不同于EOF的其他值。在标准C以及大多数最新的编译器中，如果没有错误发生，这些函数返回发送到输出流的字符数量。对于sprintf，这个计数并不包括结尾的null字符（如果发生了错误，标准C允许这些函数返回任何负值）。

C89（修正案1）指定了这些函数的3个宽字符版本。fwprintf、wprintf和swprintf。从概念上说，这些函数的输出是宽字符串，它们在控制字符串的控制之下把额外的参数转换为宽字符串。我们把这些函数称为wprintf函数家族，或简称wprintf函数群，以便把它们与原先字节定向的printf函数群进行区分。在C89修正案1中，printf和wprintf函数群的c和s转换操作均可使用长度指定符l。

C99增加了a和A转换操作，表示十六进制浮点转换，并引入了长度指定符hh、ll、j、z和t。

参考：EOF 第15.1节；十六进制浮点格式 第2.7.2节；scanf 第15.8节；stdout 第15.4节；宽字符 第2.1.4节。

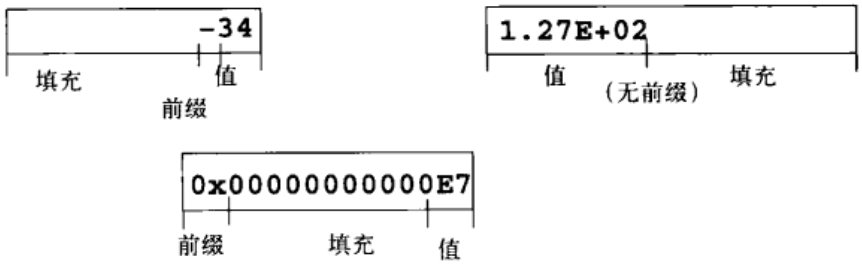
15.11.1 输出格式

控制字符串就是需要在输出中逐字复制的文本，另外还可能包含转换规范。在标准C中，控制字符串是个以起始转移状态开始和结束的多字节字符序列（不中断）。在wprintf函数群中，它是个宽字符串。

转换规范用于处理一定数量额外参数，产生一个格式化的转换操作，生成控制字符串并没有显式包含的输出字符。参数的数量必须正确，并且每个参数必须具有正确的类型，以满足控制字符串中转换规范的要求。多余的参数将被忽略，但如果参数太少，结果是不可预测的。如

果转换规范存在任何格式错误，它的效果也是不可预测的。输出转换规范与fscanf及相关函数所使用的输入转换规范相似。第15.8.2节讨论了它们之间的区别。在每个转换规范所调用的操作完成时，存在一个序列点。

一个转换规范的输出字符或宽字符序列在概念上可以划分为3种元素：前缀（如果有的话，一般是+、−或空格）；填充部分，它是空格或数字0序列，根据需要增加输出序列的宽度，以满足指定的最小值。前缀总是出现在转换后的值之前。根据转换规范，填充部分可能出现在前缀之前，把前缀与被转换的值分隔开来，也可能出现在被转换的值后面。下图显示了一些例子，闭合的方框显示了控制规范所控制的输出的范围。



15.11.2 转换规范

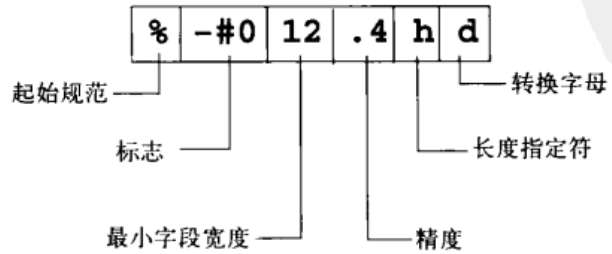
在接下来的内容中，字符、字母等术语被理解为printf函数群中的常规字符或字母（字节）以及wprintf函数群中的宽字符或字母。例如，在wsprintf中，转换规范以宽字符百分号%开始。

转换规范以百分号%开始，依次具有下面这些元素：

1. 零个或多个标志字符（−、+、0、#或空格），它们用于修改转换操作的含义。
2. 一个可选的用十进制整数常量表示的最小字段宽度。
3. 一个可选的用点号表示的精度规范，它的后面可以跟一个十进制整数。
4. 一个可选的长度指定符，可以用下列字母组合之一来表示：ll、l、L、h、hh、j或t。
5. 由单个字符表示的转换操作，取自下面这个集合：a、A、c、d、e、E、f、g、G、i、n、o、p、s、u、x、X和%。

长度指定符L和h以及转换操作i、p和n是C89所增加的。长度指定符ll、hh、j、z和t以及转换操作a和A是由C99所增加的。

转换字母出现在转换规范的最后。下面所显示的转换规范%-#012.4hd分解为几个组成元素：



15.11.3 转换标志

可选的转换标志用于修改主转换操作的含义：

-	值在字段宽度中左对齐
0	使用0而不是空格作为填充字符
+	总是产生符号，+或-
空格	总是产生符号－或空格
#	使用主转换操作的一个变型

现在详细描述这些标志字符的效果。

-标志 如果出现了负号标志，则转换后的值将在字段中左对齐，也就是说，所有的填充字符都将出现在被转换的值的右边。如果未出现负号标志，则被转换的值将在字段中右对齐。只有当显式地指定了最小字段宽度并且转换后的值小于最小宽度时，这个标志才有意义。否则，转换后的值将填充整个字段，不存在填充。

0标志 如果出现了0标志，则填充将出现在被转换值的左边，并使用0作为填充字符。只有当显式地指定了最小字段宽度并且转换后的值小于最小宽度时，0标志才有意义。在整数转换中，这个标志将被精度规范所覆盖。

如果未出现数字0标志，则使用空格作为填充字符。如果填充字符需要出现在被转换值的右边，总是使用空格作为填充字符，即使还存在-标志。

+标志 如果出现了正号标志，则有符号值的转换结果将始终具有符号，即正值的前面也会出现显式的+号（不管是否使用+标号，负值的前面总是会出现-号）。这个标志只有在转换操作a、A、d、e、E、f、g、G和i中才有效。

空格标志 如果出现了空格标志，并且有符号值的转换结果的第一个字符并不是符号(+或-)，则在转换值的前面将会添加一个空格。在左侧填充这个空格与在-标志字符的控制下出现在左侧或右侧的任何填充无关。如果一个转换规范中同时出现了空格和+标志，则空格标志被忽略，因为+标志保证了转换后的值总是以一个符号开始。这个标志只有在转换操作a、A、d、e、E、f、g、G和i中才有效。

#标志 如果出现了#标志，则使用另一种替代形式的主转换操作。这个标记只适用于转换操作a、A、d、e、E、f、g、G、i、o、x和X。#标志所提示的修改是在相关的转换操作中进行描述的。

15.11.4 最小字段宽度

转换规范可以用一个十进制整数常量表示可选的最小字段宽度。这个常量必须是个非空的并且不是以数字0开始（否则就会被当成0标志）的十进制数字序列。如果被转换的值（包括前缀）所产生的字符数量少于指定的字段宽度，那么就用填充字符填充指定宽度的值。如果被转换的值所产生的字符数量超过了指定的字段宽度，这个字段就会进行扩展，以不使用填充的方式完全容纳这些字符。

字段宽度也可以用一个星号*指定。在这种情况下，它消耗一个int类型的参数，并用它指定最小字段宽度。如果指定了负的字段宽度，其结果是不可预测的。

例子

下面这两个printf调用产生相同的输出：

```
int width=5, value;
…
printf("%5d", value);
printf("%*d", width, value);
```

15.11.5 精度

转换操作可以指定一个可选的精度规范，用一个点号以及它后面一个可选的十进制整数表示。精度规范用于控制：

1. d、i、o、u、x和X转换所打印的最小数字位数。
2. e、E和f转换中小数点右边的数字位数。
3. g和G转换中的有效数字位数。
4. s转换中从字符串所写入的最大字符数量。

如果出现了点号但没有整数，则这个整数被认为是0，它通常与省略整个精度规范具有不同的效果。

精度也可以用点号后面的一个星号来表示，在这种情况下它将消耗一个int类型的参数，并用它指定精度。如果宽度和精度字段均用星号指定，则字段宽度参数优先于精度参数。

15.11.6 长度规范

在有些转换操作中，可以出现一个可选的长度指定符，它是下面这些字母序列之一：ll（均为字母）、l（字母）、L、h、hh、j、z或t。

字母l用于转换操作d、i、o、u、x和X时，表示转换参数的类型是long或unsigned long。当它用于n转换时，它指定了参数具有long *类型。在C89中，长度指定符l还可以用于c转换，此时它指定了参数的类型是wint_t；它也可以用于s转换，此时它指定了参数的类型是wchar_t *。长度指定符l对于a、A、e、E、f、F、g和G转换无效。读者可以把这个长度指定符与L指定符进行比较，在使用它们时必须加以注意。

长度指定符ll用于转换操作d、i、o、u、x和X时，表示转换参数的类型是long long或unsigned long long。当它用于n转换时，它指定了参数具有long long *类型。ll长度指定符是C99新增的。

字母h用于转换操作d、i、o、u、x和X时，表示转换参数的类型是short或unsigned short。也就是说，尽管由于参数提升的缘故，参数会被转换为int或unsigned int类型，但它在进行转换操作之前应该转换为short或unsigned short类型。当它用于n转换时，它指定了参数具有short *类型。h长度指定符是C89新增的。

长度指定符hh用于转换操作d、i、o、u、x和X时，表示转换参数的类型是char或unsigned char。也就是说，尽管由于参数提升的缘故，参数会被转换为int或unsigned int类型，但它在进行转换操作之前应该转换为char或unsigned char类型。当它用于n转换时，它指定了参数具有signed char *类型。h长度指定符是C99新增的。

字母L用于转换操作a、A、e、E、f、F、g和G时，表示参数的类型是long double。L长度指定符是C89新增的。注意，要使用L而不是l来表示long double，因为l对于这些操作不会产生任何效果。

长度限定符j用于转换操作d、i、o、u、x和X时，表示转换参数的类型是intmax_t或uintmax_t。当它用于n转换时，它指定了参数具有intmax_t *类型。j长度指定符是C99新增的。

长度指定符z用于转换操作d、i、o、u、x和X时，表示转换参数的类型是size_t。当它用于n转换时，它指定了参数具有size_t *类型。z长度指定符是C99新增的。

长度指定符t用于转换操作d、i、o、u、x和X时，表示转换参数的类型是ptrdiff_t。当它用于n转换时，它指定了参数具有ptrdiff_t *类型。t长度指定符是C99新增的。

15.11.7 转换操作

转换操作是用单个字符表示的：a、A、c、d、e、E、f、g、G、i、n、o、p、s、u、x、X或%。 指定的转换决定了它所允许的标志和长度字符、预期的参数类型以及输出的显式形式。表15-6对这些转换操作进行了总结。我们将对每个操作进行单独讨论。

表15-6 输出转换规范

转换	预定义标志 - + # 空格	长度指定符	参数类型	默认精度[a]	输出
d，i[b]	- + # 空格	无	int	1	dd...d
		h	short		-dd...d
		l	long		+dd...d
u	- + # 空格	无	unsigned int	1	dd...d
		h	unsigned short		
		l	unsigned long		
o	- + # 空格	无	unsigned int	1	oo...o
		h	unsigned short		0oo...o
		l	unsigned long		
x，X	- + # 空格	无	unsigned int	1	hh...h
		h	unsigned short		0xhh...h
		l	unsigned long		0Xhh...h
f	- + # 空格	无	double	6	d...d.d...d
		l	double		-d...d.d...d
		L	long double		+d...d.d...d
e，E	- + # 空格	无	double	6	d.d...de+dd
		l	double		-d.d...dE-dd
		L	long double		
g，G	- + # 空格	无	double	6	和e、E或f相同
		l	double		
		L	long double		
a，A[c]	- + # 空格	无	double	6	0xh.h...hp+dd
		l	double		-0Xh.h...hP-dd
		L	long double		
c	-	无	int	1	c
		l[d]	wint_t		
s	-	无	char *	×	cc...c
		l[e]	wchar_t *		
p[b]	由编译器定义	无	void *	1	由编译器定义

（续）

转换	预定义标志 −+#空格	长度指定符	参数类型	默认精度[a]	输出
n[b]		无	int *	n/a	无
		h	short *		
		l	long *		
%		无	无	n/a	%

a 如果未指定，使用默认精度。

b C89所引入。转换i和d产生相同的输出。

c C99所引入。

d C89（修正案1）所引入。

d和i转换 它执行有符号十进制转换。如果未使用长度指定符，参数的类型应该是int。如果使用了长度指定符h，参数的类型应该是short。如果使用了长度指定符l，参数的类型应该是long。为了与fscanf兼容，标准C还保留了i操作，这是为了使输出具有统一性，它的效果与d操作相同。

经过转换的值由一个十进制数字序列组成，表示参数的绝对值。这个序列尽可能短，但不会短于指定的精度。转换后的值将根据需要加上前导的零，以满足精度规范的要求。前导的零独立于任何填充字符，后者也可能使用前导的零。如果精度是1（默认值），则转换后的值将不会使用前导0，除非参数为0，此时将输出单个0。如果精度为0并且参数为0，则输出值为空（null字符串）。

前缀是根据下面的方式计算的。如果参数为负值，前缀就是个负号。如果参数是个非负值并且指定了+标志，则前缀是个正号。如果参数是个非负值并指定了空格标志，而且没有指定+标志，则前缀是个空格。在其他情况下，前缀为空。#标志对于d和i转换没有效果。表15-7显示了d转换的一些例子。

表15-7 d转换的输出例子

示例格式	示例输出值 = 45	示例输出值 = −45
%12d	45	−45
%012d	000000000045	−00000000045
% 012d	00000000045	−00000000045
%+12d	+45	−45
%+012d	+00000000045	−00000000045
%-12d	45	−45
%- 12d	45	−45
%-+12d	+45	−45
%12.4d	0045	−0045
%-12.4d	0045	−0045

u转换 它执行无符号十进制转换。如果未使用长度指定符，参数的类型应该是unsigned。如果使用了长度指定符h，参数的类型是unsigned short。如果使用了长度指定符l，参数的类型是unsigned long。

转换后的值由一个十进制数字序列组成，它表示参数的值。这个序列将尽可能短，但不会

短于指定的精度。转换后的值将根据需要加上前导的零，以满足精度规范的要求。前导的零独立于任何填充字符，后者也可能使用前导的零。如果精度是1（默认值），则转换后的值将不会使用前导0，除非参数为0，此时将输出单个0。如果精度为0并且参数为0，则输出值为空（null字符串）。前缀总是空的。+、−和#标志对于u转换没有效果。表15-8显示了u转换的一些例子。

表15-8 u转换的输出例子

示例格式	示例输出值 = 45	示例输出值 = −45
%14u	45	4294967251
%014u	00000000000045	00004294967251
%#14u	45	4294967251
%#014u	00000000000045	00004294967251
%-14u	45	4294967251
%-#14u	45	4294967251
%14.4u	0045	4294967251
%-14.4u	0045	4294967251

o转换 它执行无符号八进制转换。如果未使用长度指定符，参数的类型应该是unsigned。如果使用了长度指定符h，参数的类型是unsigned short。如果使用了长度指定符l，参数的类型是unsigned long。

转换后的值由一个八进制数字序列组成，它表示参数的值。这个序列将尽可能短，但不会短于指定的精度。转换后的值将根据需要加上前导的零，以满足精度规范的要求。前导的零与任何填充字符独立，后者也可能使用前导的零。如果精度是1（默认值），则转换后的值将不会使用前导0，除非参数为0，此时将输出单个0。如果精度为0并且参数为0，则输出值为空（null字符串）。如果存在#标志，则前缀为0。如果不存在#标志，则前缀为空。+和−标志对于o操作无效。表15-9显示了o转换的一些例子。

表15-9 o转换的输出例子

示例格式	示例输出值 = 45	示例输出值 = −45
%14o	55	37777777723
%014o	00000000000055	00037777777723
%#14o	055	037777777723
%#014o	00000000000055	00037777777723
%-14o	55	37777777723
%-#14o	055	037777777723
%14.4o	0055	37777777723
%- 14.4o	00055	037777777723

x和X转换 它们执行无符号十六进制转换。如果未使用长度指定符，参数的类型应该是unsigned。如果使用了长度指定符h，参数的类型是unsigned short。如果使用了长度指定符l，参数的类型是unsigned long。

转换后的值由一个十六进制数字序列组成，它表示参数的值。这个序列将尽可能短，但不会短于指定的精度。x操作使用0123456789abcdef为数字，而X操作使用0123456789ABCDEF为数字。转换后的值将根据需要加上前导的零，以满足精度规范的要求。前导的零与任何填充字

符独立，后者也可能使用前导的零。如果精度是1（默认值），则转换后的值将不会使用前导0，除非参数为0，此时将输出单个0。如果精度为0并且参数为0，则输出值为空（null字符串）。如果未指定精度，则默认使用精度1。

如果存在#标志，则前缀为0x（对于x操作）或0X（对于X操作）。如果不存在#标志，则前缀为空。+和空格标志对于x和X操作无效。表15-10显示了x和X转换的一些例子。

表15-10 x和X操作的一些例子

示例格式	示例输出值 = 45	示例输出值 = −45
%12x	2d	fffffffd3
%012x	00000000002d	0000fffffffd3
%#12X	0X2D	0XFFFFFFFD3
%#012X	0X000000002D	0X00FFFFFFFD3
%-12x	2d	fffffffd3
%-#12x	0x2d	0xfffffffd3
%12.4x	002d	fffffffd3
%-#12.4x	0x002d	fffffffd3

c转换 它以字符或宽字符的形式打印参数。它将消耗1个参数。+、空格和#标志以及精度规范对于c转换操作无效。应用于参数字符的转换取决于是否存在l长度指定符以及实际所使用的是printf还是wprintf函数。表15-11列出了各种可能性。表15-12显示了c转换的一些例子。

表15-11 c转换规范

函数	长度指定符	参数类型	转换
printf	无	int	参数被转换为unsigned char，并复制到输出
	l	wint_t	参数被转换为wchar_t，就像用wcrtomb[a]转换为多字节字符一样，并输出
wprintf	无	int	参数就像调用mbtowc[m6]一样被转换为宽字符，并输出
	l	wint_t	参数被转换为wchar_t，并复制到输出

a 在字符被转换之前，wcrtomb函数的转换状态被设置为0。

表15-12 c转换的例子

示例格式	示例输出值 = '*'
%12c	*
%012c	00000000000*
%-12c	*

s转换 它以字符串的形式打印参数。它将消耗1个参数。如果不存在长度指定符l，则参数必须是个指向任何字符类型的数组的指针。如果存在长度指定符l，则参数的类型必须是wchar_t *，并指定一个宽字符序列。前缀总是为空。+、空格和#标志对于s转换无效。

如果未指定精度规范，则转换后的值是字符串参数中不包括结尾的null字符或null宽字符的字符序列。如果指定了精度规范p，则转换后的值是字符串参数的前p个字符或者是不包括结尾的null字符的字符序列（取决于哪个更短）。如果指定了精度规范，只要参数字符串包含了足够的字符，足以产生最大输出数量的字符，则它可以不必以null字符结尾。如果写入的是多字节字

符（使用长度指定符l的printf），为了防止写入不完整的多字节字符，实际写入的字节数量可以小于p。

参数字符串所发生的转换取决于是否存在长度指定符l以及实际所使用的是printf还是wprintf函数。表15-13列出了各种可能性。表15-14显示了s转换的一些例子。

表15-13 s转换规范

函数	长度指定符	参数类型	转换
printf	无	char *	字符从参数字符串被复制到输出
	l	wchar_t *	参数字符串中的宽字符就像调用了wcrtomb[a]一样被转换为多字节字符
wprintf	无	char *	参数字符串中的多字节字符就像调用了mbrtowc[a]一样被转换为宽字符
	l	wchar_t *	参数字符串中的宽字符被复制到输出

a 在第1个字符被转换之前，wcrtomb或mbrtowc函数的转换状态被设置为0。后续的转换使用前面字符所修改的状态。

表15-14 s转换的例子

示例格式	示例输出值 = “zap”	示例输出值 = “longish”
%12s	zap	longish
%12.5s	zap	longi
%012s	000000000zap	00000longish
%-12s	zap	longish

p转换 参数的类型必须是void *，并以一种由编译器所定义的格式输出。对于大多数计算机而言，它很可能与o、x或X转换所产生的格式相同。这个转换可以在标准C中找到，但并不常用。

n转换 如果未使用长度指定符，参数的类型必须是int *。如果使用了长度使用符l，参数的类型应该是long *。如果使用了长度指定符h，参数的类型应该是short *。这个操作并不输出字符，而是把到目前为止所写入的字符数量写入到指定的整数中。这个转换操作可以在标准C中找到，但并不常用。

f和F转换 它们执行有符号十进制浮点转换。它消耗1个参数。如果未使用长度指定符，参数的类型应该是double。如果使用了长度指定符L，参数的类型应该是long double。如果提供了float类型的参数，它会通过寻常参数提升被转换为double类型，因此可以用%f打印float类型的数值。

转换后的值由一个十进制数字序列组成，可能含有小数点。这个值表示参数的近似绝对值。小数点之前至少有1位数字。精度指定了出现在小数点之后的位数。如果精度为0，则小数点后面不会出现数字，而且在这种情况下小数点本身也不存在，除非还存在#标志。如果未指定精度，则精度默认为6。

如果浮点值无法准确地用这个转换所产生的数字位数所表示，则转换后的值应该是正确的浮点值四舍五入到这个数字位数的结果（有些编译器在任何情况下都不会执行正确的四舍五入）。

在C99中，如果浮点值表示无限，则使用f转换所产生的值是inf、-inf、infinity或-infinity之一（由编译器决定选择哪一个）。如果浮点值表示NaN，则使用f转换所产生的值是nan、-nan、nan(...)或-nan(...)之一，其中“...”是个由编译器所定义的字母、数字或下划线序列。F转换使用大写字母对无限和NaN进行转换。#和0标志对于无限或NaN的转换没有效果。

前缀是按下面的方式计算的。如果参数是负值，则前缀是负号。如果参数是非负值并且指定了+标志，则前缀是正号。如果参数是非负值并指定了空格标志，并且没有指定+标志，则前缀是个空格。在其他情况下，前缀为空。表15-15显示了f转换的一些例子。

表15-15 f转换的一些例子

示例格式	示例输出值 = 12.678	示例输出值 = −12.678
%10.2f	12.68	−12.68
%010.2f	000000012.68	−00000012.68
% 010.2f	00000012.68	−00000012.68
%+10.2f	+12.68	−12.68
%+010.2f	+00000012.68	−00000012.68
%−10.2f	12.68	−12.68
%−10.2f	12.68	−12.68
%−+10.4f	+12.6780	−12.6780

e和E转换 它们执行有符号十进制浮点转换。它消耗1个参数。如果未使用长度指定符，参数的类型应该是double。如果使用了长度指定符L，参数的类型应该是long double。像f转换一样，它们也允许使用float类型的参数。下面描述e转换。E转换的区别仅在于用E代替e转换中所出现的e。

转换后的值由1个十进制数字、然后可能有1个小数点和更多的十进制数字、然后是字母e、然后是个正号或负号、最后至少是2个十进制数字。除非值为0，否则字母e之前的部分表示一个1.0到9.99之间的值。字母e后面的部分是用有符号十进制整数所表示的指数值。第一部分的值乘以10的第二部分的整数次方，结果近似等于参数的绝对值。指数数字的位数对于所有的值都是相同的，是表示编译器的浮点类型的范围所需要的最大位数。表15-16显示了e和E转换的例子。

表15-16 e和E转换的一些例子

示例格式	示例输出值 = 12.678	示例输出值 = −12.678
%10.2e	1.27e+01	−1.27e+01
%010.2e	00001.27e+01	−0001.27e+01
% 010.2e	0001.27e+01	−0001.27e+01
%+10.2E	+1.27E+01	−1.27E+01
%+010.2E	+0001.27E+01	−0001.27E+01
%−10.2e	1.27e+01	−1.27e+01
%−10.2e	1.27e+01	−1.27e+01
%−+10.2e	+1.27e+01	−1.27e+01

精度指定了出现在小数点后面的数字位数。如果未提供精度，就使用默认精度6。如果精度为0，则小数点后面不会出现任何数字。而且在这种情况下小数点本身也不存在，除非还存在#标志。如果浮点值无法准确地用转换所产生的数字位数表示，则转换后的值是通过对正确的浮点值进行四舍五入而获得的。前缀的计算方式与f转换相同。无限或NaN值的转换也与f和F转换所指定的方式相同。

g和G转换 它们执行有符号十进制浮点转换。它消耗1个参数，如果未使用长度指定符，参数的类型应该是double。如果使用了长度指定符L，参数的类型应该是long double。像f转换一样，

它们也允许使用float类型的参数。下面只描述g转换。G转换和g转换的区别仅在于g使用e转换，而G使用E转换。如果指定的精度小于1，则使用精度1。如果未指定精度，则使用默认精度6。

g转换一开始与f或e转换相同，具体执行哪个取决于被转换的值。标准C规范表示只有当e转换所产生的指数小于－4或大于等于指定的精度时才使用e转换。有些编译器在指数小于－3或严格大于指定的精度时使用e转换。

转换后的值（不论是用f或e转换）将进一步修改，去掉小数点右边缀尾的0。如果处理后的结果导致小数点后面没有任何数字，则小数点也被删除。如果存在#标志，就不会执行缀尾0和小数点的清除。

前缀的计算与f和e转换相同。无限或NaN值的转换与f和F转换所指定的方式相同。

a和A转换 它们是C99新增的，执行有符号十六进制浮点转换。它消耗1个参数，如果未使用长度指定符，参数的类型应该是double。如果使用了长度指定符L，参数的类型应该是long double。和其他浮点转换一样，它也允许使用float类型的参数。下面只描述a转换，A转换与它的区别仅在于用大写字母表示十六进制数字、前缀（0X）和指数字母（P）。

转换后的值包括1个十六进制数字、然后可能是1个小数点以及更多的十六进制数字、然后是字母p、然后是个正号或负号、最后是1个或多个十进制数字。除非值为0或是非规格化的，否则前导的十六进制数字是非零的。字母p后面的部分是个二进制指数值，用有符号十进制数字表示。

精度指定了出现在小数点后面的十六进制数字的位数。如果未提供精度，则使用足够位数的数字区别double类型的值（如果FLT_RADIX为2，则默认精度足以准确地表示值）。如果精度为0，则小数点后面不会出现任何数字，而且在这种情况下小数点本身也不存在，除非还存在#标志。如果浮点值无法准确地用这个转换所产生的十六进制数字表示，则转换后的值是通过对正确的浮点值进行四舍五入而获得的。前缀的计算方式与f转换相同。

无限或NaN值的转换方式与f或F转换所指定的方式相同。

%转换 这个转换打印出一个百分号。由于百分号表示转换规范的开始，所以为了打印1个百分号，必须连续使用2个百分号。它不消耗任何参数，并且前缀为空。

标准C并不允许在%转换中出现任何标志字符、最小宽度、精度或长度指定符。完整的转换规范必须是%%。但是，其他C编译器可能会像执行其他转换操作一样执行填充。例如，在这些编译器中，转换规范%05%打印0000%。+、空格和#标志、精度规范以及长度指定符对于%转换是完全无效的。

例子

下面这个两行的程序称为“quine”，这是一个自复制的程序。当这个程序执行时，它将在标准输出中打印自身的一份拷贝（这个程序的第一行太长，无法容纳于1行之中，因此我们通过插入一个反斜杠和一个行分隔符在%cmain()后面将其断行）。

```
char*f="char*f=%c%s%c,q='%c',n='%cn',b='%c%c';%cmain()\
{printf(f,q,f,q,q,b,b,b,n,n);}%c",q='"',n='\n',b='\\';
main(){printf(f,q,f,q,q,b,b,b,n,n);}
```

下面这个单行程序几乎就是一个quine程序（我们通过插入一个反斜杠和一个行分隔符在“;main()后面将其断行，因为无法在一行中写下这个程序）。至于它为什么不完全是个quine，这

个问题就留给读者了。

```
char*f="char*f=%c%s%c;main(){printf(f,34,f,34);}";main()\
{printf(f,34,f,34);}
```

15.12 v[x]printf、v[x]scanf

语法概要

```
#include <stdarg.h>
#include <stdio.h>

int vfprintf(FILE * restrict stream,
  const char * restrict format, va_list arg);
int vprintf(
  const char * restrict format, va_list arg);
int vsprintf(char *s,
  const char * restrict format, va_list arg);
int vfscanf(FILE * restrict stream,
  const char * restrict format, va_list arg); // C99
int vscanf(
  const char * restrict format, va_list arg); // C99
int vsscanf(const char * restrict s,
  const char * restrict format, va_list arg); // C99
```

```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vfwprintf(FILE * restrict stream,
  const wchar_t * restrict format, va_list arg);
int vwprintf(
  const wchar_t * restrict format, va_list arg);
int vswprintf(wchar_t * restrict s,
  size_t n, const wchar_t * restrict format, va_list arg);
int vfwscanf(FILE * restrict stream,
  const wchar_t * restrict format, va_list arg); // C99
int vswscanf(const wchar_t * restrict s,
  const wchar_t * restrict format, va_list arg); // C99
int vwscanf(
  const wchar_t * restrict format, va_list arg); // C99
```

vfprintf、vprintf和vsprintf分别与fprintf、printf和sprintf函数相似，区别在于额外的参数是由varargs（或stdarg）工具（第11.4节）所定义的可变参数列表提供的。arg参数必须由va_start宏进行初始化，并且后面可能存在va_arg调用。如果程序员想用格式化输出工具自己定义可变参数的函数，这些函数就十分有用。函数不调用va_end工具。

C89修正案1添加了vfwprinf、vwprinf和vswprintf函数，它们分别与fwprintf、wprintf和swprintf函数相似。C99添加了对应的输入函数vfscanf、vscanf和vsscanf，以及对应的宽字符版本vfwscanf、vwscanf和vswcanf。

例子

假设想编写一个通用函数trace，打印出一个函数以及它的参数的名称。任何被追踪的函数都应该以下面这种形式的trace调用开始：

```
trace(name,format,parm1,parm2,…,parmN)
```

其中，name是被调用的函数的名称，format是个格式字符串，适用于打印参数值parm1、parm2、...、parmN。例如：

```
int f(int x, double y) /* Trace this function. */
{
    trace("f","x=%d, y=%f", x, y);
    …
}
```

下面是传统C可能采用的trace函数的实现：

```
#include <varargs.h>
#include <stdio.h>
void trace(va_alist)
    va_dcl
{
    va_list args;
    char *name;
    char *format;
    va_start(args);
    name = va_arg(args,char *);
    format = va_arg(args,char *);
    fprintf(stderr,"--> entering %s(", name);
    vfprintf(stderr, format, args);
    fprintf(stderr,")\n");
    va_end(args);
}
```

15.13 fread、fwrite

语法概要

```
#include <stdio.h>
size_t fread(
  void * restrict ptr, size_t element_size, size_t count,
  FILE * restrict stream);
size_t fwrite(
  const void * restrict ptr, size_t element_size, size_t count,
  FILE * restrict stream);
```

fread和fwrite函数分别用于二进制文件的输入和输出。在这两个函数中，stream表示输入流或输出流，ptr是个指向包含count个元素的数组的指针，每个元素的长度为element_size个字符。

fread函数从输入流中最多读取具有指定长度的count个元素到指定的数组。fread函数返回它所读取的实际元素数量，如果遇到了文件尾，它可能小于count。如果出现了错误，这个函数返回0。当函数的返回值为0时，可以使用feof或ferror工具判断是出现了错误还是遇到了文件尾。如果count或element_size为0，这个函数就不会读取任何数据，并返回0。

例子

下面这个程序读取一个包含结构类型的对象的输入文件，并打印它所读取的对象的数量。这个程序依赖exit调用关闭输入文件：

```
/* Count the number of elements
   of type "struct S" in file "in.dat" */
#include <stdio.h>
static char *FileName = "in.dat";
struct S { int a,b; double d; char str[103]; };

int main(void)
{
   struct S buffer;
   int items_read = 0;
   FILE *in_file = fopen(FileName,"r");
   if (in_file == NULL)
   { fprintf(stderr,"?Couldn't open %s\n",FileName);
     exit(1); }

   while (fread((char *) &buffer,
          sizeof(struct S), 1, in_file) == 1)
      items_read++;

   if (ferror(in_file))
      { fprintf(stderr,"?Read error, file %s record %d\n",
         FileName,items_read+1); exit(1); }
   printf("Finished; %d elements read\n",items_read);
   return 0;
}
```

fwrite函数从指定的数组中把count个长度为element_size的元素写入到流中。这个函数返回它实际写入的元素数量。除非出现了错误，否则这个返回值与count相同。

在传统C中，element_size参数的类型是unsigned，ptr参数的类型是char *，count参数的类型是int。

参考：exit 第19.3节；feof、ferror 第15.14节；fseek、ftell 第15.5节。

15.14 feof、ferror、clearerr

语法概要

```
#include <stdio.h>

int feof(FILE *stream);
int ferror(FILE *stream);
void clearerr(FILE *stream);
```

feof函数接受一个输入流作为它的参数。如果在读取输入流时检测到了文件尾，它就返回一个非零值，否则返回零。注意，即使流中不再有可以读取的字符，feof也不会认为是到达了文件尾，除非试图在最后一个字符之后进行读取。这个函数一般是在输入操作失败之后使用的。

ferror函数返回流的错误状态。如果在读取或写入到流时发生了错误，则ferror函数返回一个非零值，否则就返回0。当一个特定的流发生错误时，重复调用ferror将连续报告错误，除非显式地调用clearerr函数重置错误提示。关闭流（例如使用fclose）也将重置错误提示。

clearerr函数重置指定的流上的任何错误提示或文件尾提示，之后再调用ferror将不会报告出现了错误，除非出现了另外一个错误。

15.15 remove、rename

语法概要

```
#include <stdio.h>
int rename(
  const char *oldname, const char *newname);
int remove(const char *filename);
```

remove函数移动或删除指定的文件。如果操作成功，它就返回0，否则返回非零值。filename所指向的字符串并不会被修改。在“移动”或“删除”的实际含义上，不同的编译器可能具有不同的解释。但是对于打开已删除掉的文件的操作应该是不可能的。如果文件已打开或不存在，则remove所执行的操作是由编译器所定义的。这个函数在标准C中并不存在，一般使用UNIX所提供的一个unlink函数。

rename函数把oldname的名称修改为newname。如果操作成功，这个函数返回0，否则返回非零值。oldname和newname所指向的字符串并不会被修改。如果oldname表示一个打开的或不存在的文件，或者newname是一个已经存在的文件，则renmae所执行的操作是由编译器所定义的。

15.16 tmpfile、tmpnam、mktemp

语法概要

```
#include <stdio.h>
FILE *tmpfile(void);
char *tmpnam(char *buf);
#define L_tmpnam ...
#define TMP_MAX ...
```

tmpfile函数创建一个新文件，并使用fopen模式"w+b"（传统C中的"w+"）将它打开。如果操作成功，这个函数返回一个指向该新文件的文件指针。如果操作失败，它就返回一个null指针。调用这个函数的目的是创建一个只在程序执行期间使用的新文件。当这个文件被关闭或者当程序终止时，这个文件就会被删除。在数据写入到这个文件之后，程序员可以使用rewind函数把文件位置定位到文件的起始处，以便进行读取。

tmpnam函数用于创建不会与当前使用的其他文件冲突的新文件名。然后，程序员可以利用fopen的完全通用性用这个名称打开一个新文件。这个被创建的文件并不是“临时的”，它并不会在程序终止时被自动删除。如果buf是NULL，tmpnam返回一个指向新文件名字符串的指针，这个字符串可以被后续的tmpnam调用所更改。如果buf不是NULL，它必须指向一个不小于L_tmpnam个字符的数组。tmpnam将把这个新文件名字符串复制到这个数组，并返回buf。如果tmpnam失败，它返回一个null指针。标准C定义了TMP_MAX这个值，表示成功调用tmpnam产生唯一名称的的数量。这个值至少必须是25。

传统C的mktemp函数具有和tmpnam相同的签名，但buf（“模板”）必须指向一个带有6个缀尾的X字符的字符串，它将被其他字母或数字所改写，以形成一个唯一的文件名。这个函数返回buf的值。对mktemp的连续调用应该指定不同的模板，以保证唯一的文件名。UNIX系统常常用进程的标识来代替XXXXXX。mktemp在标准C中并不存在。

例子

在C中，一种常见但不良的编程实践是编写下面的语句：

```
ptr = fopen(mktemp("/tmp/abcXXXXXX"),"w+");
```

如果字符串常量是不可修改的，这种用法就会失败。程序员还将失去引用这个文件名字符串的能力。一种更好的办法是使用下面的语句，而且它的效率并不低。

```
char filename[]="/tmp/abcXXXXXX";
ptr = fopen(mktemp(filename),"w+");
```



第16章
基本工具

本章所描述的工具是在头文件stdlib.h中声明的。它们可以分为7种基本类型：

- 内存分配
- 随机数生成
- 数值转换和整数运算
- 环境通信
- 搜索和排序
- 多字节字符、宽字符和字符串转换

16.1 malloc、calloc、mlalloc、clalloc、free、cfree

语法概要

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t elt_count, size_t elt_size);
void *realloc(void *ptr, size_t size);
void free(void *ptr);
```

malloc函数分配一块内存，足以容纳一个大小（用sizeof操作符计算）为size的对象。这个函数返回指向这块区域的第一个元素的指针，并且无论目标对象是什么数据类型，它都保证实现了正确的对齐。然后，这个函数的调用者可以使用类型转换操作符把这个指针转换为其他指针类型。如果由于某种原因无法执行所请求的分配，这个函数就返回一个null指针。如果所请求的内存大小为0，这个函数的标准C版本要么返回一个null指针，要么返回一个不能用于访问对象的非null指针。这个函数所分配的内存并没有按照任何方式进行初始化，因此调用者无法依赖它的内容。由于malloc所分配的每块内存区域对于任何类型都已正确对齐，因此每块区域所占据的内存是对齐长度（通常是4个或8个字节）的倍数。

例子

内存函数的调用者一般把返回的指针赋值给一个适当类型的变量。在此，我们假设T是某种需要动态分配内存的对象类型，它可以是结构、数组或字符。

```
T *NewObject(void)
{
    T *objptr = (T *) malloc(sizeof(T));
    if (objptr==NULL) printf("NewObject: failed!\n");
    return objptr;
}
```

在标准C中，类型转换(T*)并不是严格必需，因为malloc返回一个void *类型的指针，并允许对objptr进行赋值时执行隐式类型转换。在传统C中，malloc的返回类型是char *，隐式转换可能会导致警告信息。为了保持C++兼容性，需要进行显式的类型转换。

calloc函数分配一块内存，这块内存足以容纳一个元素数量为elt_count、每个元素的长度为elt_size（一般由sizeof操作符所确定）的数组。这块内存被逐位清除为0，这个函数所返回的是指向这块内存的第一个元素的指针。如果由于某种原因无法执行所请求的分配，或者elt_count或elt_size为0，则返回值与malloc函数相同。注意在内存逐位清零时，浮点零或者null指针可能具有不同的表示形式。

realloc函数接受一个指针为参数，这个指针指向一块以前由标准的内存分配函数所分配的内存区域，这个函数在保留这块内存区域的内容的情况下更改它的大小。如果必要，这块内存的内容将被复制到一块新的内存区域。这个函数返回一个指向这块内存区域（可能是新分配的）的指针。如果请求无法得到满足，它就返回一个null指针，旧的内存区域并不会受到影响。如果realloc的第一个参数是个null指针，则这个函数的行为就像malloc一样。如果ptr不是null并且size为0，则realloc或者返回一个null指针，或者返回一个无法用于访问对象的指针（就像malloc一样），并且旧的内存区域被销毁。如果新的大小小于旧的大小，则旧内存区域尾部的部分旧内容将被丢弃。如果新的长度大于旧的长度，则所有的旧内容都被保留，并在尾部增加新空间。新增加的空间并不会以任何方式进行初始化，调用者必须假定它包含了垃圾信息。当realloc返回了一个与第一个参数不同的指针时，程序员应该假定旧的内存区域已经被释放，不应该再使用。

例子

下面显示了realloc的一种典型用法，对samples指针所指定的动态数组进行扩展（这种数组的元素必须通过下标表达式引用，原先指向这个数组的任何指针都将由于调用realloc而失效）。

```
#include <stdlib.h>
#define SAMPLE_INCREMENT 100
int sample_limit = 0; /* Max size of current array */
int sample_count = 0; /* Number of elements in array */
double *samples = NULL; /* Will point to array */

int AddSample( double new_sample )
    /* Add an element to the end of the array */
{
    if (sample_count < sample_limit) {
        samples[sample_count++] = new_sample;
    } else {
        /* Allocate a new, larger array. */
        int new_limit = sample_limit + SAMPLE_INCREMENT;
        double *new_array =
            realloc(samples, new_limit * sizeof(double));
        if (new_array == NULL) {
            /* Can't expand; leave samples untouched. */
            fprintf(stderr,"?AddSample: out of memory\n");
        } else {
            samples = new_array;
            sample_limit = new_limit;
            samples[sample_count++] = new_sample;
        }
    }
    return sample_count;
}
```

free函数对以前由malloc、calloc或realloc所分配的一块内存区域进行销毁。free的参数必须是与某个内存分配函数的返回值相同的指针。如果它的参数是个null指针，则调用这个函数不会

产生任何效果。当一块内存被释放时，它就不能用于任何目的。使用指向这种内存区域的指针（称为“野指针”）将产生难以预料的后果。类似，如果分配一块内存一次但多次释放它，也会产生意想不到的结果。

在内存有限的独立式环境中，程序员可能想直接控制malloc以及其他内存函数对内存的处理。这种内存通常称为堆。许多运行于独立式环境的C程序并不使用malloc，因此堆并无必要。堆的大小指定方式因编译器而异。

参考：赋值转换 第6.3.2节。

传统的内存分配工具

传统及替代工具的语法概要

```
char *malloc (unsigned size);
char *mlalloc(unsigned long size);
char *calloc (unsigned elt_count, unsigned elt_size);
char *clalloc(unsigned long elt_count, unsigned long elt_size);
void  free (char *ptr);
void  cfree(char *ptr);
char *realloc (char *ptr, unsigned size);
char *relalloc(char *ptr, unsigned long size);
```

在传统C编译器中，一般并没有头文件声明这些工具，因此程序员必须自己声明它们。

内存分配函数的size参数的最初类型为unsigned int。由于这种类型难以表示很大的内存区域，因此出现了新版本的内存分配函数，接受unsigned long类型的size参数。这些函数的返回类型是char *，并且它们所返回的指针应该显式地转换为对象指针的类型。

传统版本的free函数对以前由malloc、mlalloc、realloc或relalloc所分配的内存进行销毁。cfree函数释放以前由calloc或clalloc所分配的内存。在传统的C编译器中，向传统的free或cfree函数传递一个null指针所产生的行为因编译器而异。

16.2 rand、srand、RAND_MAX

语法概要

```
#include <stdlib.h>
int rand(void);
void srand(unsigned seed);
#define RAND_MAX …
```

连续调用rand将返回0到int类型的最大可表示的正值（含该值）之间的整数值，它们是一个伪随机数生成器所产生的连续结果。在标准C中，rand的上界是由RAND_MAX所指定的，这个值至少是32 767。

srand函数可以用于初始化伪随机数生成器，以便为rand调用提供连续的值。在调用srand之后，接下来对rand的连续调用将会产生一个特定序列的伪随机数。如果再次用同一个参数调用srand，则此后每个rand调用序列将会产生相同的伪随机数序列。在用户程序中，如果未调用srand就开始连续调用rand，它们所产生的伪随机数序列与首先调用srand(1)再连续调用rand的结果相同。

当影响程序员使用合理的伪随机数序列时，标准C的库工具将不调用rand或srand。

16.3 atof、atoi、atol、atoll

语法概要

```
#include <stdlib.h>

double     atof ( const char *str );
int        atoi ( const char *str );
long       atol ( const char *str );
long long atoll( const char *str );  // C99
```

这些函数用于转换字符串str的起始部分，它们见于许多UNIX实现中。在标准C中，为了保持兼容性，它们仍然存在，不过是根据第16.4节所描述的strtox函数而定义的，后者是更受推荐的用法。如果本节的函数无法对字符串进行转换，则它们的行为是未定义的。

除了在遇到错误时所表现的行为之外，这些函数还定义了更通用的形式，如下所示：

```
#include <stdlib.h>

double atof(const char *str) {
   return strtod(str, (char **) NULL);
}

int atoi(const char *str) {
   return (int) strtol(str, (char **) NULL, 10);
}

long atol(const char * str) {
   return strtol(str, (char **) NULL, 10);
}

long long atoll(const char * str) {
   return strtoll(str, (char **) NULL, 10);
}
```

16.4 strtod、strtof、strtold、strtol、strtoll、strtoul、strtoull

语法概要

```
#include <stdlib.h>

double strtod(
  const char * restrict str, char ** restrict ptr );
float strtof(
  const char * restrict str, char ** restrict ptr );
long double strtold(
  const char * restrict str, char ** restrict ptr );

long strtol(
  const char * restrict str, char ** restrict ptr, int base );
long long strtoll(
  const char * restrict str, char ** restrict ptr, int base );
unsigned long strtoul(
  const char * restrict str, char ** restrict ptr, int base );
unsigned long long strtoull(
  const char * restrict str, char ** restrict ptr, int base );
```

字符串—数值转换函数strtod和strtol源自UNIX System V，并被标准C所采纳。strtoul函数是C89为了完整起见而增加的。strtof、strtold、strtoll和strtoull函数则是C99新增的。一般而言，这些函数对转换提供了更灵活的控制（例如，相对sscanf而言）。C99还提供了stroto[u]imax函数（第21.8节）。

对于所有这些函数，str指向需要被转换的字符串，ptr（如果不为null）指定了一个char *类型的指针，它由这些函数所设置，指向str中被转换的字符串之后的第一个字符。如果ptr为null，它就被忽略。如果str由空白字符（由isspace函数所定义）开头，则在执行转换之后首先跳过这些空白字符。

这些函数还存在对应的宽字符版本（参见第24.4节和第21.9节）。

浮点数转换 浮点数转换函数strtod、strtof和strtold要求被转换的数由一个可选的正号或负号以及它后面1个或多个下列部分组成：

1. 十进制数字序列，可能包含1个小数点，以及后面一个可选的指数部分（由第2.7.2节所定义）。

2. 字符0x或0X，后面是一个非空的十六进制数字序列，然后是一个可选的二进制指数（由第2.7.2节所定义）。

3. 字符串INF或INFINTTY（忽略大小写）。

4. 字符串NAN或NAN(...)（忽略大小写），其中...可以是任何字母、数字或下划线字符序列。

可以与上述模式之一匹配的最长字符序列被转换为浮点数，并被返回。返回类型取决于所选择的是哪个函数。预期数值的格式与C本身的浮点数常量语法（第2.7.2节）有所不同，它可以出现可选的-或+，并且可以不需要小数点。小数点可能并不是点号（因区域而异），并且不允许出现浮点后缀（f、F、l或L）。

如果由于字符串并不匹配预期的数值模型（或为空）而无法执行转换，转换函数就返回0，并且*ptr被设置为str的值，errno被设置为ERANGE。如果转换后的数值将导致上溢，则返回值为HUGE_VAL、HUGE_VALF或HUGE_VALL（并具有正确的符号）。如果转换后的数值将导致下溢，则返回值为0。不论是上溢还是下溢，errno都被设置为ERANGE。根据这个定义，将无法区分非法的数值和导致下溢的数值，但是也许可以根据*ptr的值进行判断。当字符串与数值模型不匹配时，有些编译器可能会把errno设置为EDOM。

用strtod进行十六进制浮点数、无限和NaN进行转换是C99新增的。字符串INF和INFINITY被解释为无限。如果返回类型不能表示无限，则这些输入就被看成是导致了上溢。字符串NAN和NAN(...)表示安静的NaN。如果返回类型无法表示NaN，则这些输入就被看成是无法被转换的。

如果区域并不是"C"，这些函数可能还接受其他浮点输入格式。

整数转换 整数转换函数strtol、strtoll、strtoul和strtoull分别把参数字符串的起始部分转换为long int、long long int、unsigned long int或unsigned long long int类型的整数。预期的数值格式（因预期的基数base的值而异）在所有情况下都是相同的，并可以包含可选的－或+号，并且不允许出现整数后缀（l、L、u或U）。

如果base为0，则数值（在可选的符号之后）的格式应该是十进制常量、八进制常量或十六进制常量。数值的基数是根据它的格式推断出来的。如果base位于2和36之间（含两者），则这个数值必须由一个非零的字母和数字序列组成，表示指定基数表示形式的整数。字母a-z（或A-

Z）分别表示10-35的数字。只有小于base的那些字母允许使用。作为一种特殊情况，如果base是16，则数值可以以0x或0X开始（在符号位之后），这个前缀将被忽略。

如果无法执行转换，则这些函数返回0，*ptr被设置为str的值，errno被设置为ERANGE。如果被转换的数值将导致上溢，然这些函数返回LONG_MAX、LONG_MIN、LLONG_MAX、LLONG_MIN、ULONG_MAX或ULLONG_MAX（取决于函数的返回类型以及值的符号），errno被设置为ERANGE。

如果区域并不是"C"，这些函数可能还接受其他整数输入格式。

参考：十进制常量 第2.7节；errno 第11.2节；浮点常量 第2.7节；十六进制常量 第2.7节；HUGE_VAL 第17章；整数常量 第2.7节；isspace函数 第12.6节；LONG_MAX 、LONG_MIN、ULONG_MAX 第5.1.1节；NaN 第5.2节；八进制常量 第2.7节；类型标记 第2.7节。

16.5 abort、atexit、exit、_Exit、EXIT_FAILURE、EXIT_SUCCESS

语法概要

```
#include <stdlib.h>
#define EXIT_FAILURE ...
#define EXIT_SUCCESS ...
void exit (int status);
void _Exit(int status);                    // C99
void abort(void);
int atexit(void (*func)(void));
```

exit、_Exit和abort函数导致程序终止。控制并不会返回给这些函数的调用者。

exit函数正常地终止一个程序，并执行下面这些清理操作：

1. （仅对标准C有效）所有通过atexit函数注册的函数按照与注册顺序相反的顺序执行，执行的次数与注册的次数相同。

2. 打开的输出流被刷新。所有打开的流均被关闭。

3. 由tmpfile函数所创建的文件都被删除。

4. 程序的控制返回给宿主环境，并返回一个状态值。

按照约定，在许多系统中，status值为0表示程序成功终止，非零值表示各种异常终止。在标准C中，0这个值以及EXIT_SUCCESS宏表示成功的终止，EXIT_FAILURE宏表示不成功的终止。其他值的含义则由编译器所定义。从main函数返回一个整数值的效果相当于以这个值为参数调用exit。

_Exit函数与exit函数的不同之处在于它并不调用通过atexit注册的退出处理函数，也不调用通过signal注册的信号处理函数。至于是否执行其他清理操作（例如关闭打开的流），则由编译器所定义。_Exit是C99新增的。传统上，有些编译器用_exit这个名称提供了类似的功能。

abort函数导致程序“异常”终止。通过atexit注册的函数并不会被调用。至于abort函数是否执行清理操作，则因编译器而异。这个函数返回给宿主环境的状态值也是由编译器所定义的，但它必须表示“不成功”。在标准C以及许多传统C编译器中，对abort的调用被转换为一个可以被捕捉的特殊信号（标准C中的SIGABRT）。如果这个信号被忽略，或者它的处理函数返回，标准C编译器仍然会终止程序，但其他编译器可能会允许abort函数返回给它的调用者。断言失败

(第19.1节) 也会调用abort。

atexit函数是标准C新增的。它“注册”一个函数，使这个函数将在exit函数被调用时或者当main函数返回时被调用。当程序异常终止时（例如调用abort或raise），通过它所注册的函数并不会被调用。编译器必须至少允许程序员注册32个函数。如果注册成功，atexit函数返回0，否则返回非零值。没有办法取消一个函数的注册。在exit所执行的任何标准清理操作之前，被注册的函数按照与注册顺序相反的顺序被依次调用。每个被调用的函数不接受任何参数，并且返回类型应该是void。被注册的函数不应该试图引用任何存储类别为auto或register的对象(例如通过指针)，除非是它自己所定义的。多次注册同一个函数将导致这个函数被多次调用。有些传统C编译器用onexit这个名称实现了相似的功能。

例子

在下面这个例子中，main函数打开一个文件，并注册了cleanup函数。这个函数将在exit被调用时关闭一个文件（事实上，exit将会关闭所有文件，但程序员可能想首先关闭这个文件)。

```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
FILE *Open_File;

void cleanup(void) {
    if (Open_File != NULL) fclose(Open_File);
}

int main(void)
{
    int status;
    …
    Open_File = fopen("out.dat","w");
    status = atexit(cleanup);
    assert(status == 0);
    …
}
```

参考：assert 第19.1节；fflush 第15.2节；atexit 第19.5节；main函数 第9.9节；raise 第19.6节；return语句 第8.9节；tmpfile 第15.16节；void类型 第5.9节。

16.6 getenv

语法概要

```
#include <stdlib.h>
char * getenv( const char *name );
```

getenv函数接受一个指向字符串的指针作为它的唯一参数，这个参数按照某种因编译器而异的方式被解释为一个名称，可以由执行环境所理解。这个函数返回另一个指向字符串的指针，它表示name参数的“值”。如果指定的名称不具有值，则返回一个null指针。返回的字符串不应该被程序员所修改，但它可能被后续的getenv调用所改写。

在传统C中，(名称，值）绑定的集合也可以作为main函数的第3个非标准形参，称为env(第9.9节)。传统C常常还提供了setenv或putenv函数，可以用于设置环境变量的值。

16.7 system

语法概要

```
#include <stdlib.h>
int system( const char *command );
```

system函数把它的字符串参数传递给操作系统的命令处理器（或shell），按照某种因编译器而异的方式执行。system的行为以及返回值因编译器而异，但返回值通常是该命令的完成状态。在标准C中，system在调用时可以接受null参数。在这种情况下，如果系统未提供命令处理器，则它返回0，否则返回一个非零值。

exec

传统C语法概要

```
execl (char *name, char *argi, ..., NULL);
execlp(char *name, char *argi, ..., NULL);
execle(char *name, char *argi, ..., NULL, char *envp[]);
execv (char *name, char *argv[]);
execvp(char *name, char *argv[]);
execve(char *name, char *argv[], char *envp[]);
```

各种形式的exec并不是标准C的组成部分，它们只见于UNIX系统。不管是哪种形式的exec，它们都用文件name执行一个程序，把当前进程转换为一个新进程。它们的区别在于参数是如何提供给新进程的：

1. execl、execlp和execle函数接受可变数量的参数，其中最后一个参数必须是个null指针。按照约定，第一个参数应该与name相同，也就是说，它应该是被执行的程序的名称。

2. execv、execvp和execve函数提供了一个指向以null结尾的参数向量的指针，就像传递给main函数的参数一样。按照约定，argv[0]应该与name相同，也就是说，它应该是被执行的程序的名称。

3. execle和execve函数向新进程传递一个显式的“环境”。envp参数是个以null结尾的字符串指针向量。每个字符串的形式是"name = value"（在其他版本的exec中，调用进程的环境指针被隐式地传递给新进程）。

4. execlp和execvp函数分别与execl和execv相似，区别在于系统会在一组目录中查找文件（在正常情况下，这些目录包含了命令，这些命令通常是环境变量path或PATH的值）。

当新进程启动时，提供给exec的参数可以由新进程的main函数所使用（第9.9节）。

16.8 bsearch、qsort

语法概要

```
#include <stdlib.h>
void *bsearch(
  const void *key,
  const void *base,
  size_t count,
```

（续）

语法概要

```
    size_t size,
    int (*compar)(const void *the_key, const void *a_value));
void qsort(
    void *base,
    size_t count,
    size_t size,
    int (*compar)(const void *element1, const void *element2) );
```

bsearch函数在一个包含count个元素的数组中进行搜索，base参数指向这个数组的第一个元素。每个元素的长度是size个字符。compar是个函数，它所接受的参数是指向key的指针以及指向一个数组元素的指针。根据key是小于、等于还是大于这个数组元素，这个函数分别返回一个负值、0和正值。在开始搜索时，这个数组必须按升序（根据compar）排好了序。bsearch返回一个指向与key相匹配的数组元素的指针，如果未找到匹配的元素，它就返回一个null指针。

qsort函数对一个包含count个元素的数组进行排序，base参数指向该数组的第一个元素。每个元素的长度（字符数量）是由size所指定的。compar是个函数，它接受两个指向数组元素的指针为参数。如果第一个元素“小于”第一个元素，它就返回-1。如果第一个元素“大于”第二个元素，它就返回1。如果两个元素“相等”，它就返回0。在排序之后，这个数组将按照升序（以compar为比较标准）排序。

在这些函数内部，每次调用compar之前和之后都有一个序列点。

例子

下面这个fetch函数使用bsearch对Table进行搜索，后者是个经过排序的结构数组。key_compare函数用于测试键值。注意，fetch函数首先把这个键嵌入到一个哑元素（key_elem）中。这就允许key_compare同时适用于bsearch和qsort（第20.6节）：

```
#include <stdlib.h>
#define COUNT 100
struct elem {int key; int data; } Table[COUNT];

int key_compare(const void * e1, const void * e2)
{
    int v1 = ((struct elem *)e1)->key;
    int v2 = ((struct elem *)e2)->key;
    return (v1<v2) ? -1 : (v1>v2) ? 1 : 0;
}

int fetch(int key)
/* Return the data item associated with key in
  the table, or 0 if no such key exists. */
{
    struct elem *result;
    struct elem key_elem;
    key_elem.key = key;
    result = (struct elem *)
           bsearch(
               (void *) &key_elem, (void *) &Table[0],
               (size_t) COUNT, sizeof(struct elem),
               key_compare);
```

```
    if (result == NULL)
        return 0;
    else
        return result->data;
}
```

例子

下面这个sort_table函数使用qsort对前一个例子中的表进行排序。它使用同一个函数key_compare对表的元素进行比较：

```
void sort_table(void)
/* Sorts Table according to the key values */
{
    qsort(
        (void *)Table,
        (size_t) COUNT,
        sizeof(struct elem),
        key_compare );
}
```

传递C形式

在传统C中，bsearch和qsort函数的签名是：

```
char *bsearch(
    char *key,
    char *base,
    unsigned count,
    int size,
    int (*compar)(
        char *the_key,
        char *a_value));

void qsort(
    char *base,
    unsigned count,
    int size,
    int (*compar)(
        char *element1,
        char *element2));
```

16.9 abs、labs、llabs、div、ldiv、lldiv

语法概要

```
#include <stdlib.h>
int          abs(int           x);
long        labs(long int      x);
long long llabs(long long int x);          // C99
typedef …   div_t;
typedef …  ldiv_t;
typedef … lldiv_t;                         // C99
```

（续）

语法概要

```
div_t     div(int       n, int       d);
ldiv_t    ldiv(long     n, long      d);
lldiv_t lldiv(long long n, long long d);   // C99
```

本节所描述的函数是在标准C的stdlib.h中所定义的整型运算函数。在传统C中，它们是在math.h中定义的。abs、labs和C99新增的llabs都返回它们的参数的绝对值。它们的区别仅在于参数以及结果的类型。math.h还提供了浮点版本fabs，inttypes.h提供了最大长度的整数版本imaxabs。绝对值函数极容易实现，因此有些编译器可能会把它们当做内置的函数。标准C允许这种做法。

三个除法函数div、ldiv和C99新增的lldiv都用于计算n除以d的商和余数。它们的区别仅在于参数以及结果的类型。div_t、ldiv_t和C99新增的lldiv_t都是结构类型，它们分别包含了两个分别为int、long int和long lng int类型的成员quot和rem（顺序未指定）。返回的商quot与n/d相同，余数rem与n%d相同。当d为0或者当商或余数无法用返回类型表示时，这些函数的行为是未定义的（并不一定导致定义域错误），这是为了允许编译器以最有效率的方式实现它们。inttypes.h提供了最大长度的整数除法函数imaxdiv。

提供除法函数的原因是大多数计算机无法同时计算商和余数。因此，使用这种函数（可以实现为内联函数）比分别使用/和%速度更快。

参考：fabs 第17.2节；imaxabs 第21.7节；imaxdiv 第21.7节。

16.10 mblen、mbtowc、wctomb

语法概要

```
#include <stdlib.h>
typedef ... wchar_t;
#define MB_CUR_MAX ...
int mblen(const char *s, size_t n);
int mbtowc(wchar_t *pwc, const char *s, size_t n);
int wctomb(char *s, wchar_t wchar);
```

标准C语言需要处理区域特定的扩展字符集。在这些字符集中，每个字符无法用单个char类型的对象来表示。对于这类字符集，标准C同时提供了内部表示形式方案和外部表示形式方案。在内部，扩展字符码被认为存储在一个宽字符中，宽字符的类型是由编译器所定义的整数类型wchar_t。扩展字符串（即宽字符串）可以用wchar_t[]类型的对象表示。在外部，宽字符可以用一系列的常规字符来表示，这就是与宽字符相对应的多字节字符。读者可以参见第2.1.5节对多字节和宽字符的讨论以及第2.9节对字符集和字符编码的讨论。

本节所讨论的字符转换函数在C89修正案1中得到了增强，后者增加了新的“可重新开始”工具，包括mbrlen、btowc、wctob、mbrtowc和wcrtomb。这些新函数更为灵活，并对它的行为进行了更完整的说明。它们是在wchar.h中定义的，将在第24.2节描述。

16.10.1 编码和转换状态

本节讨论多字节字符和宽字符之间的一些转换特征，这里所讨论的术语适用于本章中的许多函数。

宽字符或多字节字符并不需要使用强制性或排他性的特殊表示形式，但是，不论是常规的字符序列还是多字节字符序列，都需要使用一个null字符'\0'表示终止符。一般而言，多字节编码依赖于状态，并通过转移字符序列来更改后续字符的含义。

在本章中，标准C最初所提供的函数将会保留它们前一次所处理的多字节字符的内部转换状态信息。修正案1提供了一种显式的类型mbstate_t，用于保存转换状态，这就允许并行处理几个字符串。但是，如果新状态参数为空，每个函数都可以使用它自己的内部状态。其他标准库函数均不允许影响这些内部转换状态。

在当前区域中，表示一个多字节字符所需要的最大字节数量是由非常量表达式MB_CUR_MAX指定的。接受一个指向多字节字符的指针s为参数的大多数函数还接受一个整数类型的参数n，后者指定了s的最大字节数量。n没有理由大于MB_CUR_MAX，但它可以小于后者，以便对转换进行限制。

假设已经有了一个当前转换状态、一个指向多字节字符的指针s以及长度n，就存在多种可能性：

1. s的前n个（或更少）字节可能形成了一个合法的多字节字符，因此对应于一个单独的宽字符wc。转换状态可以相应地更新。如果wc恰好是个null宽字符，我们就认为s产生了这个null宽字符。

2. s的所有n个字节可能形成一个合法多字节字符的起始部分，但它本身并不完整，因此无法计算对应的宽字符。在这种情况下，我们称s是不完整的多字节字符（如果n的长度至少是MB_CUR_MAX，并且s包含了冗余的转移字符，就可能发生这种情况）。

3. s的n个字节可能形成一个非法的多字节字符。也就是说，在当前的编码方案中，它们可能无法构成一个合法的或不完整的多字节字符。

更改区域的LC_CTYPE分类（第11.5节）可能会更改字符编码，使转换状态处于不确定状态。MB_CUR_MAX的值将包含足够的空间容纳转移字符。

参考：mbstate_t 第11.1节。

16.10.2 长度函数

mblen函数在s所指定的字符串中检查n个字节，以观察这些字节相对于当前转换状态而言是否表示一个合法的多字节字符。如果可以，它就返回组成这个多字节字符的字节数量。如果s非法或不完整，这个函数就返回−1这个值。如果s是个null指针，并且如果多字节字符的区域特定编码是依赖于状态的，则这个函数返回一个非零值。作为副作用，这个调用还将把所有的内部状态重置为一种预定义的“初始”条件。

16.10.3 转换为宽字符

mbtowc函数根据一个多字节字符s的内部转换状态，把它转换为一个宽字符。如果pwc不是null指针，转换结果就存储在pwc所指定的对象中。这个函数的返回值是组成这个多字节字符的

字符数量。如果s是个非法的或不完整的多字节字符，则这个函数返回−1。如果s是个null指针，并且这个多字节字符的区域特定编码依赖于状态，则mbtowc返回一个非零值。作为副作用，转换状态被重置为初始状态。

例子

下面是使用了mbtowc函数的mbstowcs函数（第16.11节）的实现：

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwcs, const char *pmbs, size_t n)
{
    size_t i = 0; /* index into output array */
    (void) mbtowc(NULL,NULL,0); /* Initial shift state */
    while (*pmbs && i < n) {
        int len = mbtowc(&pwcs[i++],pmbs,MB_CUR_MAX);
        if (len == -1) return (size_t) -1;
        pmbs += len;    /* to next multibyte character */
    }
    return i;
}
```

参考：mbstate_t 第11.1节；多字节字符 第2.1.5节；size_t 第11.1节；WEOF 第11.1节。

16.10.4 宽字符转换为多字节字符

wctomb函数把宽字符wc转换为多字节表示形式（根据它当前的转换状态），并把结果存储在s所指定的字符数组中，后者的长度至少应该是MB_CUR_MAX个字符。转换状态将被更新。它并不会在转换结果中追加一个null字符。如果wc是个合法的字符编码，这个函数返回存储在s中的字符数量，否则就返回−1。如果s是个null指针，并且如果多字节字符的区域特定编码依赖于状态，wctomb就返回一个非零值。作为副作用，这个调用还将把所有的内部状态重置为一种预定义的“初始”条件。

16.11 mbstowcs、wcstombs

语法概要

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

本节所讨论的标准C函数在宽字符串和多字节字符序列之间进行转换。这些函数的“可重新开始”版本mbsrtowcs和wcsrtombs是C89修正案1新增的，是在wchar.h中定义的，详见第24.3节。

16.11.1 转换为宽字符串

mbstowcs函数把以null结尾的字符串s中的一个多字节字符序列转换为一个对应的宽字符序列，并把结果存储在pwcs所指定的数组中。s中的多字节字符必须以初始的转换状态开始，并以一个null字符结束。每个多字节字符（包括结尾的null字符）都会被转换，就像每个都调用了mbtowc函数一样。当已经有n个元素存储到宽字符数组时，或者当到达了s的尾部（在这种情况下，就在输出中存储一个null字符），或者当发生了一个转换错误时（上述三种情况不论哪种首

先发生），转换就停止。这个函数返回它所存储的宽字符的数量（不包括结尾的null宽字符）。如果出现了转换错误，这个函数返回−1。

输出指针pwcs可能是null指针。在这种情况下，就不会存储输出的宽字符，长度参数n将被忽略。

如果已经有n个输出宽字符被写入到pwcs（并且pwcs不是null指针），输入的多字节字符串的转换将在结尾的null字符被转换之前停止。在这种情况下，src所指定的指针被设置为指向紧随最后一个被转换的多字节字符之后的位置。转换状态将被更新，它不一定是初始状态，并且函数将返回n。

如果发生了转换错误，输入的多字节字符串的转换也将永久停止。在这种情况下，src所指定的指针将更新为指向导致转换发生错误的那个多字节字符。此时这个函数返回−1（转换为size_t类型），errno的值被设置为EILSEQ，并且转换状态将处于不确定状态。

16.11.2 从宽字符串转换为多字节字符

wcstombs函数把从pwcs所指定的值开始的宽字符序列转换为一个多字节字符序列，并把结果存储在s所指定的字符数组中。每个宽字符就像调用了wctomb一样进行转换。输入的宽字符序列必须以一个null宽字符结尾。输出的多字节字符将以初始的转换状态开始。当已经有n个字符被写入到s，或者当已经到达pwcs的尾部（此时会在s的尾部添加一个null字符），或者出现了转换错误时（这三种情况无论哪种首先发生），转换就会停止。这个函数返回写入到s中的字符数量（不包括结尾的null字符）。如果出现了转换错误，这个函数就返回−1（转换为size_t类型）。

输出指针s可以是null指针。在这种情况下，不会存储任何输出字节，长度参数n被忽略。

如果已经有n个输出字符写入到s（并且s不是null指针），输入的宽字符串的转换就将停止。在这种情况下，src所指定的指针被设置为指向最后一个被转换的宽字符之后的位置。转换状态将被更新，它并不一定是初始状态，并且函数将返回n。

如果发生了转换错误，输入的宽字符串的转换也将永久停止。在这种情况下，src所指定的指针将更新为指向进行转换时导致这个错误的宽字符，并且函数返回−1（转换为size_t），errno被设置为EILSEQ，并且转换状态将处于不确定状态。

例子

下面的语句读取一个多字节字符串（mbs），并把它转换为一个宽字符串（wcs），然后把它转换为一个多字节字符串（mbs2）。如果转换函数完全填充了目标数组，我们将认为它是错误的，因为此时被转换的字符串将不是以null字符结尾的。

```
#include <stdlib.h>
#include <stdio.h>
#define MAX_WCS 100
#define MAX_MBS (100*MB_CUR_MAX)
wchar_t wcs[MAX_WCS+1];
char mbs[MAX_MBS], mbs2[MAX_MBS];
size_t len_wcs, len_mbs;

/* Read in multibyte string; check for error */
if (!fgets(mbs, MAX_MBS, stdin))
    abort();
```

```
/* Convert to wide character string; check for error */
len_wcs = mbstowcs(wcs, mbs, MAX_WCS);
if (len_wcs == MAX_WCS || len_wcs == (size_t)-1)
    abort();

/* Convert back to multibyte string; check for error */
len_mbs = wcstombs(mbs2, wcs, MAX_MBS);
if (len_mbs == MAX_MBS || len_mbs == (size_t)-1)
    abort();
```

参考：转换状态 第2.1.5节；多字节字符 第2.1.5节；宽字符 第2.1.5节。

第17章 数学函数

本章所讨论的工具是在库头文件math.h中声明的。在标准C中，stdlib.h提供了另外一些数学工具。在C99中，complex.h头文件声明了和复数有关的数学函数。

下面是本章所讨论的数学函数的一些通用规则。

参数类型 在C99之前，对浮点数进行操作的所有C库函数都被定义为只接受double类型的参数。即使实际所使用的是float类型，也不会有什么问题，因为在调用之前，float类型的参数会自动转换为double类型。现在，C99分别为float类型和long double类型的参数分别定义了一组数学函数，它们分别在原来的函数名中使用了后缀f和l。

为每种浮点类型的参数提供不同的函数名称允许程序员更灵活地控制性能和类型转换，但所付出的代价则是程序的可移植性。例如，把一个变量的类型从double修改为long double将迫使程序员修改许多函数名，否则就得忍受long double类型的参数根据double函数的签名被转换为double类型而导致的精度损失。因此，C89在头文件tgmath.h（第17.12节）中定义了一组类型通用的宏。这些宏的名称与原来double类型的库函数的名称相同，但它们会根据参数的类型调用正确的函数，就像内置的加减操作符和乘除操作符一样。如果程序员需要访问原来的函数，可以用#undef取决这些宏的定义（或者不要包含tgmath.h头文件）。C99编译器必须包含这些宏，因为在C中编写类型通用的宏是不可能的。

错误处理 数学函数可能会出现两种通用类型的错误，尽管旧式的C编译器可能并没有按照一致的方式处理它们。当一个输入参数位于函数所定义的定义域之外时，或者当一个参数具有特殊的值（例如无限或Nan）时，就会出现定义域错误。此时，errno（第11.2节）被设置为EDOM，并且函数返回一个由编译器所定义的值。0是传统的错误返回值，但有些编译器可能有更好的选择，例如特殊的“不是数”值。

如果数学函数的结果无法用该函数的返回类型来表示，则会发生值域错误。在发生这种情况时，errno应该被设置为ERANGE，并且函数返回与正确结果符号相同的最大可表示的浮点值。在C89中，这是HUGE_VAL宏的值。在C99中，还可以使用HUGE_VALF和HUGE_VALL宏。C99提供了相当大的灵活性，控制在哪些情况下表示错误，在哪些情况简单地用无限或NaN值来继续。

如果函数的结果太小，无法用返回类型来表示，则函数应该返回0。至于errno是否设置为ERANGE则由编译器决定。

17.1 abs、lbas、llabs、div、ldiv、lldiv

这些函数是在stdlib.h中定义的（参见第16.9节）。

17.2 fabs

语法概要

```
#include <math.h>

double      fabs (double x);
float       fabsf(float x); // C99
long double fabsl(long double x);   // C99
```

fabs函数返回它的参数的绝对值。整型的绝对值函数（abs、labs和llabs）是在stdlib.h中定义的。

参考：abs、labs、llabs 第16.9节；类型通用的宏 第17.12节。

17.3 ceil、floor、lrint、llrint、lround、llround、nearbyint、round、rint、trunc

语法概要

```
#include <math.h>           // All new to C99 except ceil, floor

double      ceil  (double x);
float       ceilf (float x);
long double ceill (long double x);

double      floor  (double x);
float       floorf (float x);
long double floorl (long double x);

double      nearbyint  (double x);
float       nearbyintf (float x);
long double nearbyintl (long double x);

double      rint  (double x);
float       rintf (float x);
long double rintl (long double x);

long int lrint (double x);
long int lrintf (float x);
long int lrintl (long double x);

long long int llrint  (double x);
long long int llrintf (float x);
long long int llrintl (long double x);

double      round  (double x);
float       roundf (float x);
long double roundl (long double x);

long int lround (double x);
long int lroundf (float x);
long int lroundl (long double x);

long long int llround  (double x);
long long int llroundf (float x);
long long int llroundl (long double x);

double      trunc  (double x);
float       truncf (float x);
long double truncl (long double x);
```

这些函数都用于计算与它们的浮点型参数最接近的整数。许多函数的返回值是浮点类型的，尽管实际返回的值是整数类型的，因为这些整数可能由于太大而无法用整数类型表示。除了ceil

和floor之外，本节的所有函数都是C99新增的。它们都另有类型通用的宏。这些函数的返回类型都是浮点类型，如果参数为无限（有正确的符号），则返回值也是无限。

- ceil函数返回不小于x的最小整数。
- floor函数返回不大于x的最大整数。
- round函数返回与x最接近的整数。如果x位于两个整数的正中，则round函数返回绝对值较大的那个整数（即四舍五入）。
- trunc函数返回在趋向0的方向上与x最接近的整数。对于正数，它相当于floor(x)。对于负数，它相当于ceil(x)。
- nearbyint函数根据当前的舍入方法（参见fenv.h）返回与x最接近的整数。
- lrint和llrint函数与nearbyint相似，区别在于它们把四舍五入后的值作为整数类型返回。如果四舍五入后的值无法用整数类型表示，则结果是未定义的。
- rint函数与nearbyint相似，区别在于如果结果值与参数值不同（例如，参数已不是整数），就会引发一个“不准确”浮点异常。

参考：舍入方向 第22.4节；类型通用的宏 第17.12节。

17.4 fmod、remainder、remquo

语法概要

```
#include <math.h>   // All new to C99 except fmod

double      fmod  (double x,      double y);
float       fmodf (float x, float y);
long double fmodl (long double x, long double y);

double      remainder  (double x, double y);
float       remainderf (float x, float y);
long double remainderl (long double x, long double y);

double      remquo (double x, double y, int *quo); C99
float       remquof(float x, float y, int *quo); C99
long double remquol(long double x, long double y, int *quo);
```

这些函数返回x/y的浮点余数的近似值，也就是说，对于整数n，它们返回r = x − n * y的近似数学值。它们的区别在于如何选择n，但在所有的情况下，r的绝对值小于y的绝对值。除了fmod之外，这些函数都是C99新增的，并且都另有类型通用的宏。

- fmod函数选择trunc(x/y)作为n。这意味着r的符号将与x相同。
- remainder和remquo函数选择round(x/y)作为n，区别在于如果x/y正好位于两个整数的中间，则选择那个偶整数。r的符号可能与x的符号不同。

remquo函数的返回值与remainder函数相同。另外，它们在*quo中存储一个值，后者的符号与x/y的符号相同，并且它的大小是同余模数2^k减去x/y的整数商。k是个由编译器定义的大于或等于3的整数。也就是说，*quo被设置为x/y的整数商的某几个“低端位”。因此，它可以用于一些参数衰减计算中，这个话题超出了C函数库的范畴。

如果y是0，则标准C编译器可能会产生一个定义域错误，或者使这些函数返回0。在有些旧式的编译器中，在这种情况下返回x。尽管余数在数学上是根据x/y定义的，但x/y的值并不要求

对余数定义也具有良好的表示形式。

fmod函数不应该与modf函数相混淆（第17.5节），后者从一个浮点数中提取它的小数部分和整数部分。

参考：round 第17.4节；trunc 第17.4节；类型通用的宏 第17.12节。

17.5 frexp、ldexp、modf、scalbn

语法概要

```
#include <math.h>   // All new to C99 except frexp

double      frexp (double x, int *nptr);
float       frexpf(float x, int *nptr);
long double frexpl(long double x, int *nptr);

double      ldexp (double x, int n);
float       ldexpf(float x, int n);
long double ldexpl(long double x, int n);

double      modf (double x, double *nptr);
float       modff(float x, float *nptr);
long double modfl(long double x, long double *nptr);

double      scalbn (double x, int n);
float       scalbnf(float x, int n);
long double scalbnl(long double x, int n);

double      scalbln (double x, long int n);
float       scalblnf(float x, long int n);
long double scalblnl(long double x, long int n);
```

本节的函数大部分是C99新增的，它们都另有类型通用的宏。

frexp函数把浮点数x分割为小数s和指数n，使f或者为0.0，或者$0.5 \leqslant |f| < 1.0$，并且$f * 2^n$等于x。这个函数返回小数f，作为副作用，它把指数n存储在nptr所指向的位置。如果x为0，则两个返回值均为0。如果x不是浮点数，则结果是未定义的。

ldexp函数正好与frexp函数相反，它们都计算$x * 2^n$。它可能会出现值域错误。

modf函数把一个浮点数分割为小数部分f和整数部分n，使$|f| < 1.0$，并且f + n等于x。f和n的符号与x相同。这个函数返回小数部分f，作为副作用，它把整数部分n存储在nptr所指向的位置。modf这个名称并不达意，它所计算的值更适合称为余数。modf函数不应该与fmod函数（第17.3节）混淆，后者计算两个浮点数相除的余数。据说有些旧式的C编译器以不同的方式定义modf，因此读者应该检查本地的库函数文档。在C99中，modf并不具有类型通用的宏。

scalbn和scalbln函数把浮点数x乘以b^n，对它进行扩张，其中b是FLT_RADIX。它们的计算效率要比实现计算b^n再把结果乘以x更高。这些函数可能会发生值域错误。

17.6 exp、exp2、expm1、ilogb、log、log10、log1p、log2、logb

语法概要

```
#include <math.h>     // All new in C99 except exp, log, log10

double      exp (double x);
float       expf(float x);
```

（续）

语法概要

```
long double expf(long double x);

double      exp2 (double x);
float       exp2f(float x);
long double exp2l(long double x);

double      expm1 (double x);
float       expm1f(float x);
long double expm1l(long double x);

double      log (double x);
float       logf(float x);
long double logl(long double x);

double      log10 (double x);
float       log10f(float x);
long double log10l(long double x);

double      log1p (double x);
float       log1pf(float x);
long double log1pl(long double x);

double      log2 (double x);
float       log2f(float x);
long double log2l(long double x);

int ilogb (double x);
int ilogbf(float x);
int ilogbl(long double x);
```

本节的函数大多是C99新增的，并且它们另有类型通用的宏。

exp函数计算e^x，其中e是自然对数的底。exp2函数计算2^x。expm1函数计算e^x-1（如果x在数量级上很小，则expm1(x)应该比exp(x)-1更为精确）。在所有这些函数中，如果参数过大，可能会发生值域错误。在C99之前，只存在exp函数。

log函数计算x的自然对数。log10函数计算底为10的对数，log2函数计算底为2的对数。如果x是负值，就会发生定义域错误。如果x是0或接近于0，则可能出现值域错误（趋向于负无穷），这些函数可能在不产生错误的情况下返回负无穷。有些旧式的C编译器把0当做定义域错误，并把log函数命名为ln。在C99之前，只存在log和log10函数。

logb和ilogb函数从浮点参数x的表示形式中提取指数部分。记住，字母b表示标准模式下浮点表示形式的基数，在float.h中被定义为FLT_RADIX。参数x不需要被规范化。logb函数以浮点数的形式返回（整型）指数。如果x为0，则发生定义域错误。ilogb函数以整数的形式返回指数，就像把logb的结果转换为int类型一样，但下面这几种情况例外：如果x为0，则ilogb返回FP_ILOGB；如果x为∞或−∞，则ilogb返回INT_MAX；如果x是NaN，则ilogb返回FP_ILOGBNAN。

参考：浮点模型 第5.2节；FLT_RADIX 第5.2节；类型通用的宏 第17.12节。

17.7 cbrt、fma、hypot、pow、sqrt

语法概要

```
#include <math.h>       // All new in C99 except pow, sqrt
```

（续）

语法概要

```
double      cbrt (double x);
float       cbrtf(float x);
long double cbrtl(long double x);

double      hypot (double x, double y);
float       hypotf(float x, float y);
long double hypotl(long double x, long double y);

double      fma (double x, double y, double z);
float       fmaf(float x, float y, float z);
long double fmal(long double x, long double y, long double z);

double      pow( double x, double y);
float       powf(float x, float y);
long double powl(long double x, long double y);

double      sqrt (double x);
float       sqrtf(float x);
long double sqrtl(long double x);
```

pow函数计算x^y。当x是非零值并且y是0时，结果是1.0。当x是0并且y是正数时，结果是0。如果x是负数且y不是整数，或者如果x是0且y不是正数时，就会出现定义域错误。这个函数也可能出现值域错误。

hypot函数计算$x^2 + y^2$的平方根。它们采取了有效的方法，可以避免程序员以直接的方式进行计算时可能发生的上溢和下溢，例如，sqrt (x*x+y*y)。

fma函数计算(x * y) + z。它们在计算结果时就像使用了无限精度一样，并把最终结果四舍五入为返回类型。

sqrt函数计算x的非负平方根。如果x是负数，就会出现定义域错误。

cbrt函数计算x的立方根。

参考：类型通用的宏 第17.2节。

17.8 rand、srand、RAND_MAX

这些函数是在stdlib.h中定义的（参见第16.2节）。

17.9 cos、sin、tan、cosh、sinh、tanh

语法概要

```
#include <math.h>

double      cos (double x);
float       cosf(float x); // C99
long double cosl(long double x);    // C99

double      sin (double x);
float       sinf(float x); // C99
long double sinl(long double x);    // C99

double      tan (double x);
float       tanf(float x); // C99
long double tanl(long double x);    // C99
```

（续）

语法概要

```
double      cosh (double x);
float       coshf(float x); // C99
long double coshl(long double x);   // C99

double      sinh (double x);
float       sinhf(float x); // C99
long double sinhl(long double x);   // C99

double      tanh (double x);
float       tanhf(float x); // C99
long double tanhl(long double x);   // C99
```

cos函数计算x的三角余弦函数，它是以弧度为单位的。它不会出现定义域或值域错误，但程序员应该注意到如果x的值很大，其结果就没有什么意义。

sin和tan函数分别计算三角正弦函数和正切函数。在tan函数中，如果参数接近于π/2的奇数次方，就会出现值域错误。如果参数的数量级很大，sin和tan函数的结果也没有什么意义。

cosh、sinh和tanh函数分别计算x的双曲余弦、双曲正弦和双曲正切函数。如果sinh和cosh的参数的绝对值很大，就会出现值域错误。

参考：类型通用的宏 第17.12节。

17.10 acos、asin、atan、atan2、acosh、asinh、atanh

语法概要

```
#include <math.h>   // New in C99 except acos, asin, atan, atan2

double      acos (double x);
float       acosf(float x);
long double acosl(long double x);

double      asin (double x);
float       asinf(float x);
long double asinl(long double x);

double      atan (double x);
float       atanf(float x);
long double atanl(long double x);

double      atan2(double y, double x);
float       atan2f(float y, float x);
long double atan2l(long double y, long double x);

double      acosh (double x);
float       acoshf(float x);
long double acoshl(long double x);

double      asinh (double x);
float       asinhf(float x);
long double asinhl(long double x);

double      atanh (double x);
float       atanhf(float x);
long double atanhl(long double x);
```


acos函数计算x的反余弦函数的主值。它的结果是用弧度表示的，位于0和π之间（这些函数

的值域是近似的，这是四舍五入的结果）。如果参数小于−1.0或大于1.0，就会出现定义域错误。

asin函数计算x的反正弦函数的主值。它的结果是用弧度表示的，位于−π/2和π/2之间。如果参数小于−1.0或大于1.0，就会出现定义域错误。

atan函数计算x的反正切函数的主值。它的结果是用弧度表示的，位于−π/2和π/2之间。它不会发生定义域错误或值域错误。在有些旧式的编译器中，这个函数称为arctan。

atan2函数计算y/c的反正切函数的主值。它考虑了两个参数的符号，以确定象限信息。它是根据笛卡儿坐标系统进行观察的，其结果是x的正轴到原点到（x,y）的直线之间的角度。它的结果是用弧度表示的，位于−π和π之间。如果x是0，则根据y是正数还是负数，其结果是π/2或−π/2。如果x和y都是0，则会出现定义域错误。

acosh函数计算x的反双曲余弦函数（非负）。如果x < 1，就会出现定义域错误。

asin函数计算x的反双曲正弦函数。

atan函数计算x的反双曲正切函数。如果小于−1.0或大于1.0，就会出现定义域错误。

参考：类型通用的宏 第17.11节。

17.11 fdim、fmax、fmin

语法概要

```
#include <math.h>                          // All new in C99

double      fdim (double x, double y);
float       fdimf(float x, float y);
long double fdiml(long double x, long double y);

double      fmax (double x, double y);
float       fmaxf(float x, float y);
long double fmaxl(long double x, long double y);

double      fmin (double x, double y);
float       fminf(float x, float y);
long double fminl(long double x, long double y);
```

fdim函数计算x和y之间的正差。也就是说，如果x > y，这个函数返回 x − y。如果x <= y，这个函数返回+0。

fmax函数返回两个参数中较大的那个（趋向于正无穷）。fmin函数返回两个参数中较小的那个（趋向于负无穷）。无论是哪个函数，如果一个参数是数值，另一个是NaN，就返回这个数值参数。

参考：NaN 第5.2节。类型通用的宏 第17.12节。

17.12 类型通用的宏

C99定义了一组类型通用的宏，可以提高使用了数学函数和复数函数的C程序的移植性。根据参数的类型，这些宏展开之后将调用特定的库函数。C程序可以通过包含库头文件tgmath.h来访问这些宏，这个头文件又包含了库头文件math.h和complex.h。

表17-1列出了采用原型记法的类型通用的宏。其中T表示通用类型：float、double、long double、float complex、double complex或long double complex。REAL(T)这种记法表示与复数通用类型的长度相同的实数类型。尽管大多数函数接受单个通用参数，有些函数可以接受多个通

用参数，有些函数还接受指定类型的参数（非通用）。这些参数类型将是相同的，而与通用类型无关。这张表还列出了根据参数类型实际所调用的实数和/或复数函数。这些函数是根据C库函数原先的double版本的名称，并按照一种一致性的规则命名的：复数函数加上前缀c，接受float或float complex参数的函数添加后缀f，接受long double或long double complex参数的函数则添加后缀l。

例子

编译器可以选择采用特殊的方式处理类型通用的宏。例如，sqrt宏可以按照下面的方式实现：

```
#define sqrt(x) \
  ((sizeof(x) == sizeof(float)) ? sqrt(x) :
   (sizeof(x) == sizeof(double)) ? sqrtf(x) : sqrtl(x))
```

如果以某种类型的通用参数“调用”表17-1中的一个类型通用的宏，则编译器根据下面这些规则确定应该选择调用哪个函数。在选择了函数之后，所有的参数都被转换为该函数的适当类型。如果存在函数原型，还将遵循常规的实参转换规则。

1. 如果任何通用参数的类型是long double complex，则调用long double complex版本的函数。如果不存在这样的函数，结果是未定义的。

2. 否则，如果任何通用参数的类型是double complex，则调用double complex版本的函数。如果不存在这样的函数，结果是未定义的。

3. 否则，如果任何通用参数的类型是float complex，则调用float complex版本的函数。如果不存在这样的函数，结果是未定义的。

4. 否则，如果任何通用参数的类型是long double，则调用long double版本的函数。如果不存在这样的函数，但是存在long double complex版本的函数，就调用这个复数函数。

5. 否则，如果任何通用参数的类型是double或任何通用参数为整数类型，则调用double版本的函数。如果不存在这样的函数，但是存在double complex版本的函数，就调用这个复数函数。

6. 否则，就调用float版本的函数（为了满足这个规则，所有通用参数的类型都将是float）。如果不存在这样的函数，但是存在float complex版本的函数，就调用这个复数函数。

表17-1 类型通用的宏

类型通用的宏 (tgmath.h)	实数函数 (math.h)	复数函数 (complex.h)
T acos(T x)	acos, acosf, acosl	cacos, cacosf, cacosl
T acosh(T x)	acosh, acoshf, acoshl	cacosh, cacoshf, cacoshl
T asin(T x)	asin, asinf, asinl	casin, casinf, casinl
T asinh(T x)	asinh, asinhf, asinhl	casinh, casinhf, casinhl
T atan(T x)	atan, atanf, atanl	catan, catanf, catanl
T atan2(T y, T x)	atan2, atan2f, atan2l	
T atanh(T x)	atanh, atanhf, atanhl	catanh, catanhf, catanhl
T carg(T x)		carg, cargf, cargl
T cbrt(T x)	cbrt, cbrtf, cbrtl	
T ceil(T x)	ceil, ceilf, ceill	

（续）

类型通用的宏 (tgmath.h)	实数函数 (math.h)	复数函数 (complex.h)
REAL(T) cimag(T x)		cimag, cimagf, cimagl
T conj(T x)		conj, conjf, conjl
T copysign(T x, T y)	copysign, copysignf, copysignl	
T cos(T x)	cos, cosf, cosl	ccos, ccosf, ccosl
T cosh(T x)	cosh, coshf, coshl	ccosh, ccoshf, ccoshl
T cproj(T x)		cproj, cprojf, cprojl
REAL(T) creal(T x)		creal, crealf, creall
T erf(T x)	erf, erff, erfl	
T erfc(T x)	erfc, erfcf, erfcl	
T exp(T x)	exp, expf, expl	cexp, cexpf, cexpl
T exp2(T x)	exp2, exp2f, exp2l	
T expm1(T x)	expm1, expm1f, expm1l	
T fabs(T x)	fabs, fabsf, fabsl	cabs, cabsf, cabsl
T fdim(T x, T y)	fdim, fdimf, fdiml	
T floor(T x)	floor, floorf, floorl	
T fma(T x, T y, T z)	fma, fmaf, fmal	
T fmax(T x, T y)	fmax, fmaxf, fmaxl	
T fmin(T x, T y)	fmin, fminf, fminl	
T fmod(T x, T y)	fmod, fmodf, fmodl	
T frexp(T value, int *exp)	frexp, frexpf, frexpl	
T hypot(T x, T y)	hypot, hypotf, hypotl	
int ilogb(T x)	ilogb, ilogbf, ilogbl	
T ldexp(T x, int exp)	ldexp, ldexpf, ldexpl	
T lgamma(T x)	lgamma, lgammaf, lgammal	
long long int llrint(T x)	llrint, llrintf, llrintl	
long long int llround(T x)	llround, llroundf, llroundl	
T log(T x)	log, logf, logl	clog, clogf, clogl
T log10(T x)	log10, log10f, log10l	
T log1p(T x)	log1p, log1pf, log1pl	
T log2(T x)	log2, log2f, log2l	
T logb(T x)	logb, logbf, logbl	
long int lrint(T x)	lrint, lrintf, lrintl	
long int lround(T x)	lround, lroundf, lroundl	
无	modf, modff, modfl	
T nearbyint(T x)	nearbyint, nearbyintf, nearbyintl	
T nextafter(T x)	nextafter, nextafterf, nextafterl	

（续）

类型通用的宏 (tgmath.h)	实数函数 (math.h)	复数函数 (complex.h)
T nexttoward(T x, long double y)	nexttoward, nexttowardf, nexttowardl	
T pow(T x, T y)	pow, powf, powl	cpow, cpowf, cpowl
T remainder(T x, T y)	remainder, remainderf, remainderl	
T remquo(T x, Ty, int *quo)	remquo, remquof, remquol	
T rint(T x)	rint, rintf, rintl	
T round(T x)	round, roundf, roundl	
T scalbln(T x, long int n)	scalbln, scalblnf, scalblnl	
T scalbn(T x, int n)	scalbn, scalbnf, scalbnl	
T sin(T x)	sin, sinf, sinl	csin, csinf, csinl
T sinh(T x)	sinh, sinhf, sinhl	csinh, csinhf, csinhl
T sqrt(T x)	sqrt, sqrtf, sqrtl	csqrt, csqrtf, csqrtl
T tan(T x)	tan, tanf, tanl	ctan, ctanf, ctanl
T tanh(T x)	tanh, tanhf, tanhl	ctanh, ctanhf, ctanhl
T tgamma(T x)	tgamma, tgammaf, tgammal	
T trunc(T x)	trunc, truncf, truncl	

17.13 erf、erfc、lgamma、tgamma

语法概要

```
#include <math.h>          // All new in C99 but comon in UNIX
double      erf (double x);
float       erff(float x);
long double erfl(long double x);
double      erfc (double x);
float       erfcf(float x);
long double erfcl(long double x);
double      lgamma (double x);
float       lgammaf(float x);
long double lgammal(long double x);
double      tgamma (double x);
float       tgammaf(float x);
long double tgammal(long double x);
```

erf函数用于计算错误函数

$$\frac{2}{\sqrt{\pi}}\cdot\int_0^x e^{-t^2}\,\mathrm{d}t$$

erfc函数计算1 − erf(x)，相当于

$$\frac{2}{\sqrt{\pi}}\cdot\int_{x}^{\infty}e^{-t^2}\,dt$$

lgamma函数计算大小为x的gamma函数的自然对数：

$$\log|\Gamma(x)|$$

tgamma函数计算x的gamma函数：

$$\Gamma(x)$$

17.14 fpclassify、isfinite、isinf、isnan、isnormal、signbit

语法概要

```
#include <math.h>          // All new in C99
int fpclassify(real-floating-type x);
#define FP_INFINITE ...
#define FP_NAN ...
#define FP_NORMAL ...
#define FP_SUBNORMAL ...
#define FP_ZERO ...
int isfinite(real-floating-type x);
int isinf(real-floating-type x);
int isnan(real-floating-type x);
int isnormal(real-floating-type x);
int signbit(real-floating-type x);
```

在本节所讨论的宏中，如果参数被列为real-floating-type（实数浮点类型），则它是类型通用的宏，它们的参数可以是任何浮点实数类型的表达式。由于浮点表达式在求值时可以使用比它们的实际“语义”更高的精度，因此这些宏在检查参数的类型之前必须把它们转换为正确的类型表示形式。正如C标准所指出的那样，在long double格式中规格化的数在double格式中可能是非规格化的，在float格式中可能变成0。

fpclassify宏返回FP_INFINITE、FP_NAN、FP_NORMAL、FP_SUBNORMAL或FP_ZERO这几个值之一。在这些宏中，每个宏都是一个不同的整数常量表达式。C编译器也可能指定其他以FP_和一个大写字母开头的用于分类的宏。

isfinite宏只有当它的参数既不是无限又不是NaN时才返回一个非零值。非规格化的数都是有限的。

isinf宏只有当它的参数是无限（不论正负）时才返回一个非零值。

isnan宏只有当它的参数是NaN时才返回一个非零值。

isnormal宏只有当它的参数是个规格化的数时才返回一个非零值。如果它的参数是0、非规格化数、无限或NaN值，这个宏返回0。

signbit宏只有当它的参数是负数时才返回一个非零值。

17.15 copysign、nan、nextafter、nexttoward

语法概要

```
#include <math.h>      // All new in C99

double      copysign (double x, double y);
float       copysignf(float x, float y);
long double copysignl(long double x, long double y);

double      nan (const char *tagp);
float       nanf(const char *tagp);
long double nanl(const char *tagp);

double      nextafter (double x, double y);
float       nextafterf(float x, float y);
long double nextafterl(long double x, long double y);

double      nexttoward (double x, long double y);
float       nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);
```

本节所讨论的函数对浮点值进行操作。

copysign函数返回符号与y相同的x值。

nan函数返回一个“安静”的NaN值。如果C编译器提供了安静的NaN值，它的内容是由tagp所指定的字符串提示的，否则nan就返回0。下面的调用：

```
nan("char-sequence")
nan("")
nan(NULL)
```

分别相当于下面这些调用：

```
strtod("NAN(char-sequence)", (char **) NULL)
strtod("NAN()", (char **) NULL)
strtod("NAN", (char **) NULL)
```

对nanf和nanl的调用被映射到对应的strtof和strftodl调用。

nextafter函数返回y方向上趋向x的下一个可以表示的浮点值。如果不存在这样的有限值，就会出现值域错误。如果x和y相等，则这个函数返回y。注意，实参和返回值实际上都被转换为形式参数和返回类型，即使是在宏实现中，因为准确的浮点表示形式是非常重要的。

nexttoward函数与nextafter函数相似，区别在于y的类型总是long double。

参考：安静的NaN 第5.2节；strtod 第13.8节。

17.16 isgreater、isgreaterequal、isless、islessequal、islessgreater、isunordered

语法概要

```
#include <math.h>          // All new in C99
int isgreater(real-floating-type x, real-floating-type y);
int isgreaterequal(real-floating-type x, real-floating-type y);
```

（续）

语法概要
`int isless(real-floating-type x, real-floating-type y);`
`int islessequal(real-floating-type x, real-floating-type y);`
`int islessgreater(real-floating-type x, real-floating-type y);`
`int isunordered(real-floating-type x, real-floating-type y);`

如果两个浮点值有任何一个为NaN，则它们是无序的。正常情况下，对无序的值使用C的比较操作将会发生“非法的”浮点异常。本节所讨论的类型通用的比较宏将不会引发异常，因此适用于有些类型的浮点密集型编程。如果C编译器在使用比较操作符时不会引发非法异常，则这些操作符的行为与这些宏相同。

isunordered宏只有当它的参数无序时才返回真。

isgreater宏只有当它的参数无序时才返回0，否则返回(x) > (y)的值。

isgreaterequal宏只有当它的参数无序时才返回0，否则返回(x) >= (y)的值。

isless宏只有当它的参数无序时才返回0，否则返回(x) < (y)的值。

islessequal宏只有当它的参数无序时才返回0，否则返回(x) <= (y)的值。

islessgreater宏只有当它的参数无序时才返回0，否则返回(x) < (y) || (x) > (y)的值（它的参数不会两次求值）。

参考：NaN 第5.2节。

第18章 时间和日期函数



本章所讨论的工具向程序员提供了提取和使用（日历）日期、时间和进程时间（即运行中的程序所使用的处理器时间）的方法。

日历时间可以记录程序的运行日期或者文件的写入日期，或者计算过去或将来的日期。日历时间是用两种形式表示的：时间函数所返回的简单算术值以及根据gmtime和localtime函数所返回的值计算产生的分解结构形式。标准C的strftime函数提供了区域特定的时间格式。

进程时间常常用于测量一个程序或程序的某个部分的执行速度。进程时间是clock函数所返回的一个算术值（通常是整数类型）。

18.1 clock、clock_t、CLOCKS_PER_SEC、times

语法概要

```
#include <time.h>
typedef ... clock_t;
#define CLOCKS_PER_SEC …
clock_t clock(void);
```

clock函数返回当前进程所使用的处理器时间的近似值。时间的单位取决于机器，通常是毫秒。标准C版本的clock函数允许编译器实现者选择使用任何算术类型clock_t来表示进程时间。每秒时间单位的数量（“时钟滴答”）是由CLOCKS_PER_SEC宏所定义的。如果处理器时间不可访问，这个函数就返回−1（也是clock_t类型）。

程序员应该注意进程时间的“回绕”。例如，如果clock_t类型是用32位表示的，并且clock函数返回用毫秒表示的时间，这个时间在经过36分钟后会“回绕”到它的起始值。

例子

下面是使用clock函数对一个标准C程序进行计时的例子：

```
#include <time.h>
clock_t start, finish;
…
start = clock();
process();
finish = clock();
printf("process() took %f seconds to execute\n",
       ((double) (finish - start)) / CLOCKS_PER_SEC );
```

转换为double类型允许clock_t和CLOCKS_PER_SEC为浮点数或整数。

在传统C中，clock的返回类型是long，但返回值实际上是unsigned long类型。long的使用是在unsigned long类型出现在C语言中之前。在计算进程时间时，总是应该使用无符号算术类型。在有些非标准的编译器中，可以看到times函数，它代替了clock函数。这个函数返回一个结构化的值，报告进程时间的各个成员，每个成员一般是用1/60秒为单位测量的。这个函数的签名是：

```
#include <sys/types.h>
#include <sys/times.h>
long clock(void);
void times(struct tms *);
struct tms { … };
```

例子

下面使用非标准的times函数粗略地模拟标准C的clock函数的功能：

```
#include <sys/types.h>
#include <sys/times.h>
#define CLOCKS_PER_SEC 60
long clock(void)
{
    struct tms tmsbuf;
    times(&tmsbuf);
    return (tmsbuf.tms_utime + tmsbuf.tms_stime);
}
```

前面这个结构使用了一种time_t类型，它表示“进程时间”单位，因此与标准C所定义的“日历时间”类型time_t并不相同。

参考：time 第18.2节；time_t 第18.2节。

18.2 time、time_t

语法概要

```
#include <time.h>
typedef ... time_t;
time_t time(time_t *tptr);
```

标准C的time函数返回用一个time_t类型的值封装的当前日历时间。time_t可以是任何算术类型。如果形参tptr并不是null指针，则返回值还将存储在*tptr中。如果遇到了错误，这个函数就返回-1（转换为time_t类型）。

一般情况下，time所返回的值被传递给asctime或ctime函数，转换为一种可读的格式；或者传递给localtime或gmtime函数，转换为一种更容易处理的格式。计算两个日历时间的间隔可以用标准C的difftime函数来完成。在其他编译器中，程序员必须处理gmtime所产生的分解时间，或者必须依赖时间的自定义表示形式，即从过去的一个任意日期开始所流逝的秒数（通常是从1970年1月1日起）。

传统编译器所使用的是long而不是time_t，但这个函数的返回值在逻辑上是unsigned long。当发生错误时，这个函数返回-1L。在UNIX System V中，errno也会被设置为EFAULT。

参考：asctime 第18.3节；ctime 第18.3节；difftime 第18.5节；errno 第11.2节；gmtime 第18.4节；localtime 第18.4节。

18.3 asctime、ctime

语法概要

```
#include <time.h>
```

（续）

语法概要

```
char *asctime( const struct tm *ts );
char *ctime( const time_t *timptr );
```

asctime和ctime函数都返回一个指向字符串的指针，表示一个可打印的日期和时间，形式如下：

```
"Sat May 15 17:30:00 1982\n"
```

asctime函数接受一个指向结构化日历时间的指针为它的唯一参数。这个结构是localtime或gmtime函数根据time所返回的算术值产生的。ctime函数接受一个指向time所返回的值的指针为参数，因此ctime(tp)相当于asctime(localtime(tp))。

在大多数编译器（包括许多遵循标准C的编译器）中，这些函数返回一个指向静态数据区域的指针，因此再次调用这些函数之前，应该复制或打印（使用strcpy函数）返回字符串的内容。

在传统C中，这两个函数所使用的是long类型而不是time_t，这些函数可以在头文件sys/time.h中找到。

例子

许多程序需要打印当前日期和时间。下面显示了如何使用time和ctime函数来完成这个任务：

```
#include <time.h>
#include <stdio.h>
time_t now;
...
now = time(NULL);
printf("The current date and time is: %s",ctime(&now));
```

参考：gmtime 第18.4节；localtime 第18.4节；strcpy 第13.3节；struct tm 第18.4节；time 第18.2节。

18.4 gmtime、localtime、mktime

语法概要

```
#include <time.h>
struct tm { ... };
struct tm *gmtime( const time_t *t );
struct tm *localtime( const time_t *t );
time_t mktime( struct tm *tmptr );
```

gmtime和localtime函数把time函数所返回的算术日历时间转换为一种struct tm类型的“分解”形式。gmtime转换为格林尼治统一时间（GMT），而localtime转换为本地时间，并考虑了时区以及可能的夏令时的因素。如果遇到了错误，这两个函数都返回一个null指针。这两个函数在UNIX系统和标准C之间是可以移植的。struct tm结构包含了表18-1所列出的字段。所有的字段都是int类型。

在大多数编译器（包括许多标准C编译器）中，gmtime和localtime返回一个指针，它指向一个在每次调用时都将被改写的静态数据区域。因此，在再次调用这些函数之前，应该使用或复制前一次调用所返回的结构。

表18-1 struct tm类型的字段

名称	单位	范围
tm_sec	分钟之后的秒数	0..61*
tm_min	小时之后的分钟数	0..59
tm_hour	午夜以来的小时数	0..23
tm_mday	当月的第几天	1..31
tm_mon	自1月以来的月份	0..11
tm_year	自1900年以来的年份	
tm_wday	从星期天以来的天数	0..6
tm_yday	从1月1日以来的天数	0..365
tm_isdst	夏令时标志	如果是夏令制，值>0；如果不是，值为0；如果不清楚，值<0

mktime函数（标准C）在tmptr参数所指定的分解形式的本地时间的基础上构建一个time_t类型的值。tmptr->wday和tmptr->yday的值被mktime所忽略。如果调用成功，mktime返回一个新的时间值，并调整*tmptr的内容，设置它的tm_wday和tm_yday成员。如果指定的日历时间无法用一个time_t类型的值表示，则mktime函数返回−1（转换为time_t）。第18.5节显示了一个例子。

在传统C中，时间函数的签名是：

```
#include <sys/time.h>
struct tm { … };
struct tm *gmtime(long *t);
struct tm *localtime(long *t);
```

18.5 difftime

语法概要

```
#include <time.h>
double difftime( time_t t1, time_t t0 );
```

difftime函数只见于标准C。它从日历时间t1中减去日历时间t0，以double类型的形式返回两个时间相差的秒数。程序员无法保证time_t所封装的日历时间是标量值（例如微秒的数量），因此必须使用difftime而不是简单地把两个time_t类型的值相减。

例子

下面这个函数返回1990年4月15日和当前时间之间的秒数。

```
#include <time.h>
…
double Secs_Since_Apr_15(void)
{
    struct tm Apr_15_struct = {0}; /* Set all fields to 0 */
    time_t Apr_15_t;
    Apr_15_struct.tm_year = 90;
    Apr_15_struct.tm_mon  = 3;
    Apr_15_struct.tm_mday = 15;
    Apr_15_t = mktime(&Apr_15_struct);
    if (Apr_15_t == (time_t)-1)
        return 0.0; /* error */
```

```
    else
        return difftime( time(NULL), Apr_15_t);
}
```

参考：time_t 第18.2节。

18.6 strtime、wcsftime

语法概要

```
#include <time.h>
size_t strftime(
  char *s, size_t maxsize,
  const char *format,
  const struct tm *timeptr);
```

```
#include <wchar.h>
size_t wcsftime(
  wchar_t *s, size_t maxsize,
  const wchar_t *format,
  const struct tm *timeptr);
```

这些函数只能在标准C中找到。和sprintf（第15.11节）一样，strftime在多字节字符串format的控制下把字符存储到参数s所指定的字符数组中。但是，strftime只对timeptr（第18.4节）所指定的单个日期和时间量进行格式化，通过format进行格式化的方式也不同于sprintf。在s所指定的数组中，不会存储超过maxsize个字符（包括结尾的null字符）。这个函数返回它实际存储的字符数量（不包括结尾的null字符）。如果maxsize不够大，无法容纳整个格式化字符串，则这个函数返回0，输出字符串的内容是未定义的。strftime的格式是区域特定的，取决于LC_TIM分类的设置（参见第20.1节setlocale）。

. C89修正案1添加了wcsftime函数，把日期和时间格式化为一个宽字符串。这个函数与wsprintf函数（第15.11节）类似。

格式化代码

format字符串由转换规范和其他多字节字符的任意混合方式组成。在格式化处理时，转换规范被其他字符所代替，如表18-2所示。其他多字节字符则简单复制到输出。转换规范由%字符、可选的多个限定符字母E或O（大写字母）之一以及指定了转换的单个字符组成。

表18-2 strftime的格式化代码

Ltr	被什么所取代	所使用的timeptr字段
a	简写的星期几名称。在"C"区域中，它总是%A的前3个字符，例如"Mon"（等）	tm_wday
A	完整的星期几名称。在"C"区域中，如“Monday”（等）.	tm_wday
b	简写的月名。在"C"区域中，它总是%B的前3个字符，例如"Fe"（等）	tm_mon
B	完整的月名。在"C"区域中，如"February"（等）	tm_mon
c	区域特定的日期和时间，在"C"区域中，它与%a %b %e %T %Y相同	任一或全部
C	（C99）年的最后两位数字（00-99）	tm_year

（续）

Ltr	被什么所取代	所使用的timeptr字段
d	以十进制整数表示的某月的几日（01-31）	tm_mday
D	与%m / %d / %y相同	tm_mon、tm_mday、tm_year
e	某月的几日（1-31），个位数的日加空格前缀	tm_mday
F	ISO 8601日期格式：%Y - %m - %d	tm_mon、tm_mday、tm_year
g	基于星期的年的最后两位数字（00-99）[a]	tm_year、tm_wday、tm_yday
G	基于星期的年（0000-9999）	tm_year、tm_wday、tm_yday
h	与%b相同	tm_mon
H	用十进制整数表示的小时（00-23，24小时制）	tm_hour
I	用十进制整数表示的小时（00-11，12小时制）	tm_hour
j	用十进制整数表示的某年的几日（001-366）	tm_yday
m	用十进制整数表示的月份（01-12）	tm_mon
M	用十进制整数表示的分钟数（00-59）	tm_min
n	（C99）被一个换行符所代替	无
p	与12小时制的区域AM/PM指示符相同。在"C"区域中，它是AM或PM	tm_hour
r	（C99）区域的12小时时钟时间。在"C"区域中，它是%I:%M:%S %p	tm_hour、tm_min、tm_sec
R	（C99）与%H:%M相同	tm_hour、tm_min
S	用十进制整数表示的秒数（00-60）[b]	tm_sec
t	（C99）被一个水平制表符所代替	无
T	（C99）ISO 8601时间格式：%H:%M:%S	tm_hour、tm_min、tm_sec
u	（C99）ISO 8601星期几编号（1-7），星期一为1	tm_wday
U	当年的第几星期（00-53）[c]	tm_year、tm_wday、tm_yday
V	（C99）ISO 8601基于星期的年的第几星期（01-53）	tm_year、tm_wday、tm_yday
w	用十进制整数表示的星期几（0-6，星期天为0）	tm_wday
W	年的第几星期（00-53）[d]	tm_year、tm_wday、tm_wday
x	区域特定的日期。在"C"区域中，它是%m / %d / %y	任一或全部
X	区域特定的时间。在"C"区域中，它是%T	任一或全部
y	年的最后两位数字（00-99）	tm_year
Y	以十进制整数表示的世纪年（例如1952）	tm_year
z	（C99）ISO 8601与UTC的时区偏移值，或者无。-530表示比Greenwich晚5小时30分钟（在它的西边）	tm_isdst
Z	时区名称或它的简写，如果时区未知，则为空。在"C"区域中，它是由编译器所定义的	tm_isdst
%	单个%符号	无

a 参见正文中关于“基于星期的年”的定义。

b 允许闰秒（60）。

c 星期序号1包含了第1个星期天，以前的日子为第0星期。

d 星期序号1包含了第1个星期一，以前的日子为第0星期。

C99新增了限定符和其他许多转换字母。限定符E可以作用于转换字母c、C、x、X、y和Y，它指定了使用区域的替代表示形式（未指定）。限定符O可以作用于转换字母d、e、H、I、M、m、s、u、U、V、w、W和y，它指定了使用区域的替代数值符号（未指定）。在"C"区域中，这些限定符被忽略。

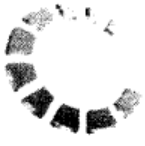
有些C99转换字母根据ISO 8601基于星期的年指定转换。在这个系统中，每星期是从星期一

开始的，每年的第1星期包含了1月4日（或者说，第1星期包含了新年的至少4天）。这意味着1月1日、2日和3日可能是前一年的最后一星期，12月29、30和31日可能是下一年的第1个星期。例如，1999年1月2日星期六是1998年的第53星期。读者可以把它与%U和%W进行对比，后者会根据需要引入一个不完整的“第0星期”。

例子

下面显示了使用strftime函数实现第18.3节的asctime的一种可行的方法。由于格式化是区域特定的，所以不易预测asctime的输出字符串的长度（包括结尾的null字符）：

```
#include <time.h>
#define TIME_SIZE 80  /* hope this is big enough */
char *asctime2( const struct tm *tm )
{
    static char time_buffer[TIME_SIZE];
    size_t len;
    len = strftime( time_buffer, TIME_SIZE,
        "%a %b %d %H:%M:%S %Y\n", tm);
    if (len == 0)
        return NULL;  /* time_buffer is too short */
    else
        return time_buffer;
}
```



第19章 控制函数

本章所讨论的工具提供了C程序的标准控制流的扩展。它们是由头文件assert.h、setjmp.h和signal.h提供的。在本书以前版本的本章中所描述的一些控制函数被转移到了第16章，包括system函数和exit相关的函数。

19.1 assert、NDEBUG

语法概要

```
#include <assert.h>

#ifndef NDEBUG
void assert( int expression );
#else
#define assert(x) ((void)0)
#endif
```

assert宏接受任何类型的整数作为它的唯一参数（许多编译器允许使用任何标量类型）。如果这个值为0并且未定义NDEBUG宏，则assert将在标准输出流中打印一条诊断信息，并调用abort（在标准C中）或exit（在传统C中）终止程序。assert工具总是被实现为宏。为了在源文件中使用这个工具，必须包含头文件assert.h。诊断信息将包括参数的文本、文件名（__FILE__）和行号（__LINE__）。C99编译器也可以使用函数名（__func__）。

如果在读取头文件assert.h时已经定义了NDEBUG宏，通常把assert定义为空语句，将这个工具禁用。此时不会打印诊断信息，assert的参数并不会进行求值。

例子

assert宏一般是在程序开发期间使用的，验证某些条件在运行时是否为真。它向程序的阅读者提供了可靠的文档，对于程序的调试很有帮助。当一个程序可以投入使用时，可以很方便地禁用断言，在此之后它们就不会产生任何运行时开销。在下面这个例子中，断言比普通的文档更为合适，因为后者可能会被误解：

```
#include <assert.h>
int f(int x)
{
    /* x should be between 1 and 10 */    /* !? */
    assert(x>0 && x<10);
    …
}
```

参考：abort 第19.3节；exit 第19.3节；__func__ 第2.6.1节；__LINE__ 第3.3.4节。

19.2 system、exec

参见第16.7节。

19.3 exit、abort

参见第16.5节。

19.4 setjmp、longjmp、jmp_buf

语法概要

```
#include <setjmp.h>
typedef ... jmp_buf;
int setjmp( jmp_buf env );
void longjmp( jmp_buf env, int status );
```

setjmp和longjmp函数实现了一种原始形式的非本地跳转，可以用于处理异常情况。这种工具一般被认为比signal（第19.6节）更具移植性，但后者现在也是标准C的一部分。

setjmp宏在“跳转缓冲区”env（一个由编译器所定义的数组）中记录它的调用者的环境，并向调用者返回0（jmp_buf类型必须实现为数组类型，这样实际传递给setjmp的是个指向env的指针）。

longjmp函数所接受的参数包括一个以前通过调用setjmp所填充的跳转缓冲区以及一个整数值status（通常为非零值）。调用longjmp的效果是使程序再次从setjmp的调用地点返回，但此时的返回值是status。有些编译器（包括标准C）并不允许longjmp使setjmp返回0。如果调用longjmp时status为0，则从setjmp返回1。

setjmp和longjmp函数在难以实现方面是臭名昭著的，程序员应该可能少地对它们作出假设。当setjmp返回一个非零值时，程序员可以认为longjmp被调用时静态变量具有正确的值。在标准C中，对于包含setjmp调用的那个函数所声明的自动局部变量，只有当它们是volatile限定的类型或者它们的值在最初调用setjmp直到对应的longjmp调用之间不会发生变化时才能保证具有正确的值。而且，标准C要求setjmp调用或者是条完整的表达式语句（可能转换为void），或者出现在一条简单赋值语句的右侧，或者作为if、switch、do、while或for语句的控制表达式使用，并采用下面的形式之一：

```
(setjmp(...))
(!setjmp(...))
(exp relop setjmp(...))
(setjmp(...) relop exp)
```

其中，exp是个整数常量表达式，relop是个关系或相等操作符。C89要求longjmp在非嵌套的信号（中断）处理函数中能够正确地执行操作。但是，在有些旧式的编译器中，在中断处理期间或信号处理期间调用setjmp或longjmp将无法保证它们的操作是正确的。

如果longjmp的跳转缓冲区参数并不是由setjmp所设置的，或者包含setjmp的函数在调用longjmp之前就已终止，它的行为就是未定义的。

例子

```
#include <setjmp.h>
jmp_buf ErrorEnv;
...
```

```
int guard(void)
    /* Return 0 if successful; else longjmp code. */
{
    int status = setjmp(ErrorEnv);
    if ( status != 0) return status;  /* error */
    process();
    return 0;
}

int process(void)
{   …
    if (error_happened) longjmp(ErrorEnv, error_code);
    …
}
```

longjmp函数是在process函数遇到错误时被调用的。guard函数是“回退点”，即longjmp将把控制转移到这个地点。程序员应该直接或间接地在guard函数中调用process函数。这就保证了longjmp不会在guard函数返回之后才调用，也不会有任何操作依赖于包含longjmp的process函数的局部变量的值（这是一种保守的策略）。注意，必须对setjmp的返回值进行测试，判断是否由于longjmp所返回。

19.5 atexit

参见第16.5节。

19.6 signal、raise、gsignal、ssignal、psignal

语法概要

```
#include <signal.h>
#define SIG_IGN …
#define SIG_DFL …
#define SIG_ERR …
#define SIGxxx …
...
void (*signal( int sig, void (*func)(int) )) (int);
int raise( int sig );
typedef … sig_atomic_t;

/* Non-Standard extensions: */
int kill( int pid, intsig );
int (*ssignal( int softsig, int (*func)(int) )) (int);
int gsignal( int softsig );
void psignal( int sig, char *prefix );
```

信号是（潜在的）异步事件，可能需要用户程序或操作系统进行特殊的处理。信号是用整数值进行命名的，每个编译器在signal.h头文件中定义了一组信号，均以字母SIG开头。信号可能由计算机的错误检测机制、由用户程序通过kill或raise、或者程序的外部活动所触发或引发。ssignal和psignal函数所使用的软件信号是用户定义的，它们的值一般位于1-15的范围之内。在其他情况下，它们就像常规的信号一样进行操作。

sig信号的信号处理函数是一个当sig信号被“引发”时所调用的用户函数。程序期望这个处理函数执行一些有用的操作并返回，一般使程序在中断点恢复。处理函数也可以调用exit。信号处理函数是常规的C函数，它接受一个参数，即被引发的信号：

```
void my_handler(int the_signal) { … }
```

有些非标准的编译器在遇到某些预定义的信号时会向信号处理函数传递一些额外的参数。

signal函数用于把信号处理函数与特定的信号相关联。在正常情况下，signal所接受的参数是一个信号值和一个指向该信号的处理函数的指针。如果关联成功，signal函数返回指向前一个信号处理函数的指针，否则就返回−1（标准C中为SIG_ERR），并设置errno。

例子

```
void new_handler(int sig) { … }
void (*old_handler)();
…
/* Set new handler, saving old handler */
old_handler = signal( sig, &new_handler );
if (old_handler==SIG_ERR)
   fprintf(stderr,"?Couldn't establish new handler.\n");
…
/* Restore old handler */
if (signal(sig,old_handler)==SIG_ERR)
   printf("?Couldn't put back old handler.\n");
```

signal函数的参数和返回值可以是两个特殊值SIG_IGN和SIG_DFL。以signal(sig, SIG_IGN)的形式调用signal意味着sig信号将被忽略。以signal(sig, SIG_DFL)的形式调用signal意味着sig信号接受它的“默认”处理，通常意味着有些信号将被忽略，而另一些信号将会终止程序。

ssignal函数（见于UNIX System V）的工作方式与signal相似，但它只用于与gsignal协同工作，处理用户定义的软件信号。提供给ssignal的处理函数所返回的整数值将成为gsignal函数的返回值。

raise和gsignal在当前的进程中引发指定的信号（或软件信号）。kill函数在指定的进程中引发指定的信号，它的移植性较差。

当一个信号被引发并且该信号已经通过signal或gsignal函数关联了一个信号处理函数时，这个信号处理函数就接管了控制。标准C（以及其他大多数编译器）或者在信号处理函数接管控制前把相关联的信号处理函数重置为SIG_DEF，或者以其他方式阻塞该信号。这是为了预防不必要的递归（由于一些历史和性能原因，SIGILL信号是否会发生这种情况是由编译器所定义的）。信号处理函数可能会返回，在这种情况下程序的执行将会在中断点继续，但有下面这些注意事项：

1. 如果信号是由raise或gsignal所引发的，则这些函数返回到它们的调用者。

2. 如果信号是由abort所引发的，则标准C程序将会终止。其他编译器可能会返回到abort的调用者。

3. 如果被处理的信号是SIGFPE或其他由编译器所定义的计算信号，则返回的行为是未定义的。

信号处理函数应该避免调用除signal之外的其他库函数，因为有些信号可能是由于库函数而引发的，而库函数（除了signal之外）并不保证可以再次进入。

标准C定义了表19-1所列出的宏，它们表示一些标准信号。这些信号在许多C编译器中都是

十分常见的。

表19-1 标准信号

宏名	信号含义
SIGABRT	异常终止，例如由于调用abort工具所致
SIGFPE	错误的算术运算，例如试图除以零
SIGILL	由非法的计算机指令所导致的错误
SIGINT	注意信号，例如由于交互式用户点击了一个特殊的键所引发
SIGSEGV	非法访问内存
SIGTERM	用户或其他程序所引发的终止信号

psignal函数（不是标准C）在标准错误输出中打印字符串prefix（按惯例为程序的名称）以及sig信号的简短描述。这个函数可以在那些打算调用exit或abort的信号处理函数中使用。

参考：exit 第19.3节；longjmp 第19.4节。

19.7 sleep、alarm

语法概要

```
void sleep( unsigned seconds );
unsigned alarm( unsigned seconds );
```

这些函数并不是标准C的一部分。alarm函数把一个内部系统时钟设置为指定的秒数，并返回这个计时器以前的秒数。当计时器到期时，程序中就会引发SIGALRM信号。如果alarm函数的参数为0，则调用的效果相当于取消以前所有的alarm请求。alarm函数可以用于从各种类型的死锁中解脱出来。

sleep函数使程序暂停指定的秒数，然后sleep函数返回并恢复程序的执行。sleep一般是使用与alarm相同的计时器实现的。如果睡眠时间超过了alarm计时器所设置的时间，sleep函数将会在SIGALRM信号被处理之后立即返回。如果睡眠时间小于alarm计时器所设置的时间，则sleep函数将在返回之际重置这个计时器，使它能够接收到预期的SIGALRM信号。

在处理了所有信号之后，编译器通常会终止sleep函数。有些编译器提供了未睡眠的秒数作为sleep函数的返回值（unsigned类型）。

有些编译器可能会把这些函数定义为接受unsigned long类型的参数。

参考：signal 第19.6节。

第20章 区 域



标准C在设计时面向国际社区，它的成员可能使用不同的字母，并在格式化数字、货币数量、日期和时间上具有不同的约定。C语言标准允许编译器相应地调整运行时函数库的行为，并保持不同国家之间的合理移植性。

国家、文化和语言的约定集合称为区域（locale），和区域有关的工具是在头文件locale.h中定义的。区域所影响的东西包括小数点和货币数量的格式、字母和校正序列（例如第12章的字符处理工具）以及日期和时间值的格式。在运行时，可以通过在一组由编译器所定义的区域集合中进行选择，对“当前区域”进行更改。标准C只定义了“C”区域，它指定了与C的最初定义一致的最小环境。

20.1 setlocale

语法概要

```
#include <locale.h>
#define LC_ALL ...
#define LC_COLLATE ...
#define LC_CTYPE ...
#define LC_MONETARY ...
#define LC_NUMERIC ...
#define LC_TIME ...
char *setlocale( int category, const char *locale );
```

setlocale函数用于更改运行时函数库的区域特定的特性。第一个参数category是个代码，表示需要被更改的行为。category参数允许使用的值包括表20-1所定义的宏的值，并且可能包括编译器所定义的以LC_开头的其他分类。

表20-1 预定义的setlocale分类

名称	受影响的行为
LC_ALL	所有行为
LC_COLLATE	strcoll和strxfrm工具的行为
LC_CTYPE	字符处理函数（第12章）
LC_MONETARY	localeconv所返回的货币信息
LC_NUMERIC	localeconv所返回的小数点和非货币信息
LC_TIME	strftime工具的行为

第二个参数locale是个由编译器所定义的字符串，表示一个区域的名称，这个区域的约定将作用于category所指定的行为。locale的唯一预定义值是表示标准C区域的"C"。如果它是空字符串""，表示它是由编译器所定义的本地区域。运行时函数库在默认情况下使用C区域，除非显式地调用setlocale函数更改了当前区域。

如果setlocale函数的locale参数是个null指针，这个函数就不会更改当前区域，但是返回一个

字符串指针，表示当前区域中指定分类的名称。如果以后调用setlocale时用同一个category值以及这个调用的返回字符串作为locale参数的值，其效果相当于把区域的行为修改为以null为参数调用setlocale时所生效的行为。例如，如果程序员打算修改区域的特定行为，可以首先以LC_ALL和NULL为参数调用setlocale，获取当前区域的一个值，以便在将来恢复到以前的区域特定行为。程序员不能修改它所返回的字符串，它可能会被后续的setlocale调用所改写。

如果setlocale的locale参数并不是null，setlocale就会修改当前区域，并返回一个表示这个新区域的字符串。如果由于任何原因setlocale无法满足这个请求，它就返回一个null指针。程序员不能修改它所返回的字符串，它可能会被后续的setlocale调用所改写。

例子

下面的original_locale函数返回当前区域的描述，以后可以根据需要恢复它。setlocale所返回的字符串并不存在最大长度，因此必须动态分配它的空间。

```
#include <locale.h>
#include <string.h>
#include <stdlib.h>
char *original_locale(void)
{
    char *temp, *copy;
    temp = setlocale(LC_ALL, NULL);
    if (temp == NULL) return NULL;  /* setlocale() failed */
    copy = (char *)malloc(strlen(temp)+1);
    if (copy == NULL) return NULL;  /* malloc() failed */
    strcpy(copy,temp);
    return copy;
}
```

下面的代码使用original_locale修改并恢复区域行为：

```
#include <locale.h>
extern char *original_locale(void);
char *saved_locale;
…
saved_locale = original_locale();
setlocale(LC_ALL,"");           /* Change to native locale */
setlocale(LC_ALL,saved_locale);/* Restore former locale   */
```

参考：malloc 第16.1节；localeconv 第20.2节；strcoll 第13.10节；strcpy 第13.3节；strftime 第18.6节；strlen 第13.4节；strxfrm 第13.10节。

20.2 localeconv

语法概要

```
#include <locale.h>
struct lconv {…};
struct lconv *localeconv(void);
```

localeconv函数用于获取当前区域中及格式化数字及货币数量约定有关的信息。这就允许程序员实现一些应用程序特定的转换和格式化函数，并且它们在不同的区域间具有移植性，从而避免了向标准C添加区域特定的转换工具的必要性。localeconv函数返回一个指向struct lconv类

型的对象的指针，这个对象必须包含表20-2所示的成员。程序员不能修改这个函数所返回的结构，它可能会被后续的localeconv调用所改写。在struct lconv中，值为空字符串的字符串成员以及值为CHAR_MAX的字符成员应该被理解为“不知道”。

例子

下面这个函数使用localeconv函数，用正确的小数点字符打印出一个浮点值：

```
#include <locale.h>
#include <stdio.h>
...
void P(int int_part, int fract_part, int fract_digits)
{
    struct lconv *lconv = localeconv();
    char *pt = lconv->decimal_point;
    /* If *pt is the empty string, use "." */
    if (!*pt) pt = ".";
    printf("%d%s%0*d\n",
        int_part, pt, fract_digits, fract_part);
}
```

表20-2列出了struct lconv的其他内容，下面将对它们进行讨论。

表20-2 lconv结构的成员

类型	名称	用途	C区域中的值
char *	decimal_point	小数点字符（非货币）	"."
char *	thousands_sep	非货币数字分组分隔字符	""
char *	grouping	非货币数字分组	""
char *	int_curr_symbol	三字符国际货币系统，加上分隔国际符号和货币数量的字符	""
char *	currency_symbol	当前区域的本地货币符号	""
char *	mon_decimal_point	小数点字符（货币）	""
char *	mon_thousands_sep	货币数字分组分隔符	""
char *	mon_grouping	货币数字分组	""
char *	positive_sign	非负货币数量的符号字符	""
char *	negative_sign	负货币数量的符号字符	""
char	int_frac_digits	国际货币格式小数点右边所显示的数字	CHAR_MAX
char	frac_digits	非国际货币格式小数点右边所显示的数字	CHAR_MAX
char	p_cs_precedes	如果currency_symbol出现在非负货币值之前，值为1；如果在后面，则值为0	CHAR_MAX
char	p_sep_by_space	如果currency_symbol与非负货币值用一个空格分隔，则值为1，否则为0	CHAR_MAX
char	n_cs_precedes	类似用于负值的p_cs_precedes	CHAR_MAX
char	p_sep_by_space	类似用于负值的p_sep_by_space	CHAR_MAX
char	p_sign_posn	非负货币数量（加上它的currency_symbol）的positive_sign的位置	CHAR_MAX
char	n_sign_posn	负货币数量（加上它的currency_symbol）的negative_sign的位置	CHAR_MAX

数字分组 struct lconv的grouping和mon_grouping成员是char类型的整数序列。尽管它们被描述为字符串，但这个字符串只是对一些小整数序列进行编码而已。这个序列中的每个整数指定了一组中的数字位数。第一个整数对应于小数点左边的第一组，第二个整数对应于再往左的下一组，接下来以此类推。整数0（字符串末尾的null字符）表示重复前一个数字组。整数CHAR_MAX表

示不再进行更多的分组。常规的千分位分组是由"\3"所指定的，第一组的3位数字延续到后续各组中，字符串"\1\2\3\127"把1234567890分组为1234 567 89 0（假定CHAR_MAX为127）。

符号位置 struct lconv的p_sign_posn和n_sign_posn成员分别决定了在什么地方替换positive_sign和negative_sign。它们可以采用的值以及这些值的含义是：

0 数值两边加上括号和currency_symbol。

1 数值前加上符号字符串和currency_symbol。

2 符号字符串出现在数值和currency_symbol之后。

3 符号字符串就在currency_symbol之前。

4 符号字符串紧随currency_symbol之后。

表20-3和表20-4显示了货币格式化的完整例子，它们都取自标准C标准。表20-3显示了四个国家的典型货币格式化。表20-4显示了指表20-3的格式时所使用的struct lconv的成员的值。

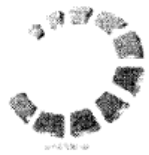
表20-3 格式化货币数量的例子

国家	格式		
	正值	负值	国际
意大利	L.1.234	-L.1.234	ITL.1.234
荷兰	F 1.234,56	F -1.234,56	NLG 1.234,56
挪威	kr1.234,56	kr1.234,56-	NOK 1.234,56
瑞士	SFrs.1.234.56	SFrs.1,234.56C	CHF 1,234.56

表20-4 lconv结构内容的例子

成员	意大利	荷兰	挪威	瑞士
int_curr_symbol	"ITL."	"NLG "	"NOK "	"CHF "
currency_symbol	"L."	"F"	"kr"	"SFrs."
mon_decimal_point	""	","	","	"."
mon_thousands_sep	"."	"."	"."	","
mon_grouping	"\3"	"\3"	"\3"	"\3"
positive_sign	""	""	""	""
negative_sign	"-"	"-"	"-"	"C"
int_frac_digits	0	2	2	2
frac_digits	0	2	2	2
p_cs_precedes	1	1	1	1
p_sep_by_space	0	1	0	0
n_cs_precedes	1	1	1	1
p_sep_by_space	0	1	0	0
p_sign_posn	1	1	1	1
n_sign_posn	1	4	2	2

第21章 扩展整数类型



本章所讨论的C99工具提供了一些具有各种不同特性的附加整数类型声明。这些工具是在头文件stdint.h和inttypes.h中提供的。stdint.h头文件包含了特定长度的整数类型的基本定义，它们在宿主式和独立式环境中都是必须实现的。inttypes.h头文件包含了stdint.h，并添加了一些可移植的格式化和转换函数。它只在宿主式环境是必须实现的。

“C的精神”是让标准类型的长度选择由编译器所决定。遗憾的是，这使得编写可移植代码的任务变得困难。本章的工具可以解决移植性问题，但这两个头文件所包含的定义数量之多可能令人吃惊。

参考：宿主式和独立式环境 第1.4节。

21.1 基本规则

这些函数库都包含了大量的类型、宏和函数，它们都是以一种常规的风格创建的。本节讨论作用于函数库的基本规则。

21.1.1 类型的种类

函数库包含了一些不同“种类”的整数类型和宏，其中有些用宽度为N的类型进行了参数化。N必须是个没有前导零的无符号十进制整数，表示一个类型的宽度（以位为单位）。

例子

正好为8位的整数类型称为int8_t和uint8_t（而不是int08_t和uint08_t）。速度最快的整数类型至少为8位，被命名为int_fast8_t和uint_fast8_t。“固定长度”和“最快”是两种不同“种类”的类型。

21.1.2 要么都定义，要么都不定义

在有些情况下，由编译器决定定义哪些类型（即N的哪些值）。但是，如果定义了某个N值的一种特定种类的类型，就必须定义这个种类和长度的类型的有符号和无符号版本，以及所有相关的宏。如果一个特定种类和长度的类型是可选的，并且编译器选择了不定义它，则编译器也不应该定义与它相关联的任何类型或宏。

例子

如果编译器定义了固定的长度为16的整数类型，就必须定义int16-t和uint16-t类型以及所有相关的INT16_MIN、INT16_MAX、UINT16_MAX、PRId16、PRIi16、PRIo16、PRIu16、PRIx16、PRIX16、SCNd16、SCNi16、SCNo16、SCNu16和SCNx16宏。如果编译器并没有定义固定的长度为16位的整数类型，则这些类型和宏都不应该被定义。

21.1.3 MIN和MAX限制

...MIN和...MAX宏指定了被定义类型可以表示的最小值和最大值，定义了该类型的范围，就

像limits.h中的...MIN和...MAX宏定义了标准类型的范围一样。在大多数情况下，范围的最小数量级是由C99所指定的。

例子

int16_t和uint16_t是固定长度的16位整数类型。它们的范围是：

```
#define INT16_MIN -32768
#define INT16_MAX 32767
#define UINT16_MAX 65535
```

参考：limits.h 表5-2。

21.1.4 PRI...和SCN...格式字符串宏

PRIcKN和SCNcKN宏分别是printf和scanf函数群的格式控制字符串。c表示一种特定的转换操作符字母：d、i、o、u、x或X。K表示类型的种类：空或LEAST、FAST、PTR或MAX。N是用位表示的宽度。表21-1列出了这些宏的完整集合。

表21-1 整数类型的格式控制字符串宏（*N*=以位为单位的类型宽度）

	固定长度种类	最小长度种类	快速长度类型	指针种类	最大值种类
有符号printf格式	PRId*N*	PRIdLEAST*N*	PRIdFAST*N*	PRIdPTR	PRIdMAX
	PRIi*N*	PRIiLEAST*N*	PRIiFAST*N*	PRIiPTR	PRIiMAX
无符号printf格式	PRIo*N*	PRIoLEAST*N*	PRIoFAST*N*	PRIoPTR	PRIoMAX
	PRIu*N*	PRIuLEAST*N*	PRIuFAST*N*	PRIuPTR	PRIuMAX
	PRIx*N*	PRIxLEAST*N*	PRIxFAST*N*	PRIxPTR	PRIxMAX
	PRIX*N*	PRIXLEAST*N*	PRIXFAST*N*	PRIXPTR	PRIXMAX
有符号scanf格式	SCNd*N*	SCNdLEAST*N*	SCNdFAST*N*	SCNdPTR	SCNdMAX
	SCNi*N*	SCNiLEAST*N*	SCNiFAST*N*	SCNiPTR	SCNiMAX
无符号scanf格式	SCNo*N*	SCNoLEAST*N*	SCNoFAST*N*	SCNoPTR	SCNoMAX
	SCNu*N*	SCNuLEAST*N*	SCNuFAST*N*	SCNuPTR	SCNuMAX
	SCNx*N*	SCNxLEAST*N*	SCNxFAST*N*	SCNxPTR	SCNxMAX

PRI...宏被展开为包含printf转换操作字符c（d、i、o、u、x或X）的字符串常量，前面有个可选的长度指定符，适合该特定种类和长度的类型的输出值。类似，SCN...宏被展开为包含scanf转换操作字符c（d、i、o、u或x）的字符串常量，前面有个可选的长度指定符，适合对数值输入进行转换，并把它存储在由指向该特定种类和长度的类型的指针所指定的对象中。

例子

最小整数的宽度至少是64位，它们被命名为int_least64_t和uint_least64_t（种类K是LEAST）。如果这些类型分别被定义为long和unsigned long，则可以预期在inttypes.h中找到下面这些定义：

```
#define PRIdLEAST64 "ld"
#define PRIiLEAST64 "li"
#define PRIoLEAST64 "lo"
#define PRIuLEAST64 "lu"
#define PRIxLEAST64 "lx"
#define PRIXLEAST64 "lX"
#define SCNdLEAST64 "ld"
#define SCNiLEAST64 "li"
#define SCNoLEAST64 "lo"
```

```
#define SCNuLEAST64 "lu"
#define SCNxLEAST64 "lx"
```

现在，假设变量a的类型是long，b的类型是int_least64_t。下面这两条语句显示了打印这些值的两种方式。第2种方式的移植性更好，因为它和int_least64_t的实际类型无关。

```
printf("a=%25ld\n", a); /* usual */
printf("b=%25" PRIdLEAST64 "\n", b); /* portable */
```

参考：limits.h 表5-2；printf转换 第15.11.17节；scanf转换 第15.8.2节。

21.2 固定长度的整数类型

语法概要

```
#include <stdint.h>                    // All C99
typedef ... intN_t
typedef ... uintN_t
#define  INTN_MIN  -2^(N-1)
#define  INTN_MAX  2^(N-1)-1
#define UINTN_MAX  2^N-1

#include <inttypes.h>
#define PRIcN "..."
#define SCNcN "..."
```

这些类型和宏定义了具有固定长度并且没有填充位的整数类型。...MIN和...MAX宏必须具有如上所示的准确值。

这些类型在stdint.h中是可选的。但是，如果编译器定义固定宽度8、16、32或64位的整数类型，则必须定义对应的类型和宏。编译器还可以定义其他固定宽度的整数类型。

例子

下面这些定义是许多使用字节编码的计算机上的编译器所预期的：

```
#include <limits.h> /* SCHAR_MIN, SCHAR_MAX, UCHAR_MAX */
typedef signed char              int8_t;
typedef unsigned char uint8_t;
typedef short int16_t;
typedef unsigned short         uint16_t;
typedef int int32_t;
typedef unsigned int           uint32_t;
typedef long long int int64_t;
typedef unsigned long long int uint64_t;
#define INT8_MIN SCHAR_MIN
#define INT8_MAX SCHAR_MAX
#define UINT8_MAX UCHAR_MAX
#define PRId8 "hhd"
#define SCNo64 "llo"
// etc.
```

当计算机的字长将来扩大时，我们可以预计long被命名为int64_t，long long int被命名为int128_t。

21.3 具有最小宽度的最小长度类型

语法概要

```
#include <stdint.h>                    // All C99
typedef ...  int_leastN_t
typedef ... uint_leastN_t
#define  INT_LEASTN_MIN  -(2^(N-1)-1)
#define  INT_LEASTN_MAX  2^(N-1)-1
#define UINT_LEASTN_MAX  2^N-1
#define  INTN_C(constant)  ...
#define UINTN_C(constant)  ...

#include <inttypes.h>
#define PRIcLEASTN "..."
#define SCNcLEASTN "..."
```

这些类型和宏定义了具有某个最小长度的最小整数类型。...MIN和...MAX宏必须具有相同的符号，并至少具有上面所显示之值的数量级。由于这些类型必须是具有指定宽度的最小类型，因此如果某个特定的N存在一种固定的宽度类型（第21.2节），则这种固定宽度类型也必定是同一个N值的最小长度类型。

所有的C99编译器都必须定义N = 8、16、32和64的这些类型和宏。其他N值的定义是可选的，但是如果提供了其他N值，则与这个N值相关联的所有类型和宏都必须定义。

例子

一种在32位字长的计算机上使用的编译器可能把char、short和int都定义为32位的类型。在这种情况下，固定宽度类型int8_t和int16_t（以及对应的无符号类型）将不会被定义，而最小宽度类型int8_t和int16_t则必须被定义为32位类型之一，例如int。

INTN_C宏接受一个十进制、十六进制或八进制的常量为参数，并展开为一个值相同的int_leastN_t类型的有符号整数常量。UINTN_C宏被展开为一个uint_leastN_t类型的无符号整数常量。这些宏会在常量后面加上适当的后缀。

例子

如果int_least64_t被定义为long long int，则INT64_C(1)将是1LL，UINT54_C(1)则是1ULL。

21.4 具有最小宽度的快速类型

语法概要

```
#include <stdint.h>                    // All C99
typedef ...  int_fastN_t
typedef ... uint_fastN_t
#define  INT_FASTN_MIN  -(2^(N-1)-1)
#define  INT_FASTN_MAX  2^(N-1)-1
#define UINT_FASTN_MAX  2^N-1

#include <inttypes.h>
#define PRIcFASTN "..."
#define SCNcFASTN "..."
```

这些类型和宏定义了具有某个最小长度的最快速的整数类型。...MIN和...MAX宏必须具有相同的符号，并且它们的值必须具有如上所示的数量级。所有的C99编译器必须为N = 8、16、32和64定义这些类型和宏。其他N值的定义是可选的。但是如果提供了任何其他N值，则与这个N值相关联的所有类型和宏都必须定义。

确定哪种类型“最快”可能需要一个判断调用，这是编译器实现者的任务，并且它对于一种类型所有可能的用法并不一定都正确。例如，标量运算的最快类型和访问数组元素的最快类型可能并不相同。

例子

在一台对32位的运算进行了优化的采用字节编址的计算机上，C编译器可能选择推荐32位字节的类型，即使实际所需要的位数更少。下面是一组可能出现的stdint.h定义集合。这个例子只显示了有符号整数类型。

```
typedef char int8_t;
typedef char int_least8_t;
typedef int int_fast8_t;

typedef short int16_t;
typedef short int_least16_t;
typedef int int_fast16_t;

typedef int int32_t;
typedef int int_least32_t;
typedef int     int_fast32_t;
```

21.5 指针长度和最大长度整数类型

语法概要

```
#include <stdint.h>                    // All C99
typedef ... intptr_t;
typedef ... uintptr_t;
#define  INTPTR_MIN  -(2^15-1)
#define  INTPTR_MAX  2^15-1
#define UINTPTR_MA X 2^16-1

typedef ...  intmax_t;
typedef ... uintmax_t;
#define  INTMAX_MIN  -(2^63-1)
#define  INTMAX_MAX  2^63-1
#define UINTMAX_MA X 2^64-1
#define  INTMAX_C(constant) ...
#define UINTMAX_C(constant) ...

#include <inttypes.h>
#define PRIcPTR "..."
#define SCNcPTR "..."
#define PRIcMAX "..."
#define SCNcMAX "..."
```

intptr_t类型和uintptr_t类型分别是可以存储任何对象指针的有符号和无符号整数类型。也就是说，如果P是个void *类型的值，P可以转换为intptr_t或uintptr_t类型，然后再转换回void *类型，

其结果仍然是原来的指针P。...MIN和...MAX宏必须具有相同的符号，并至少具有与上面所示的值相同的数量级。这些类型是可选的，因为可能不存在这样的整数类型（但这种情况极少见）。

intmax_t和uintmax_t类型分别是编译器所定义的最长有符号和无符号整数类型。所有的C编译器都必须定义这两种类型。由于C99编译器允许提供扩展的整数类型，因此intmax_t类型可能并不是标准C类型之一，例如long long int。...MIN和...MAX宏必须具有相同的符号，并且具有与上面所显示的值相同的数量级。

INTMAX_C宏接受一个十进制、十六进制或八进制的常量为参数，并展开为一个具有相同值的intmax_t类型的整数常量。UINTMAX_C宏展开为一个uintmax_t类型的整数常量。

参考：limits.h 表5-2。

21.6 ptrdiff_t、size_t、wchar_t、wint_t和sig_atomic_t的范围

语法概要

```
#include <stdint.h>
#define PTRDIFF_MIN     ...              // All C99
#define PTRDIFF_MAX     ...
#define SIZE_MAX        ...
#define WCHAR_MIN       ...
#define WCHAR_MAX       ...
#define WINT_MIN        ...
#define WINT_MAX        ...
#define SIG_ATOMIC_MIN  ...
#define SIG_ATOMIC_MAX  ...
```

本节所讨论的宏展开后的结果是预处理器常量表达式，它们表示stddef.h和wchar.h所定义的各种类型的数字范围。所有的编译器都必须定义它们。

PTRDIFF_MIN和PTRDIFF_MAX指定了ptrdiff_t类型的范围，它至少必须是个16位的有符号类型。

SIZE_MAX指定了size_t类型可以表示的最大值。

WCHAR_MIN和WCHAR_MAX指定了wchar_t的范围，它们可以是至少8位的有符号或无符号类型。

WINT_MIN和WINT_MAX指定了wint_t的范围，它们可以是至少16位的有符号或无符号类型。

SIG_ATOMIC_MIN和SIG_ATOMIC_MAX指定了sig_atomic_t的范围，它们可以是至少8位的有符号或无符号类型。

参考：ptrdif_t 第11.1节；sig_atomic_t 第19.6节；size_t 第11.1节；wchar_t 第24.1节；wint_t 第24.1节。

21.7 imaxabs、imaxdiv、imaxdiv_t

语法概要

```
#include <inttypes.h>
typedef ... imaxdiv_t;                          // All C99
intmax_t imaxabs( intmax_t x );
imaxdiv_t imaxdiv( intmax_t n, intmax_t d );
```

本节所讨论的工具支持最长整数类型上的基本运算，它们与stdlib.h所定义的abs和div函数相似。imaxabs函数计算它的参数的绝对值。如果参数的绝对值无法表示，其结果是未定义的。

imaxdiv函数在一个操作中同时计算n/d和n%d，计算结果分别存储在imaxdiv_t结构类型的quot和rem成员中。imaxdiv_t中的成员顺序并没有明确规定。

参考：abs 第16.9节；div 第16.9节。

21.8 strtoimax、strtouimax

语法概要

```
#include <inttypes.h>
intmax_t strtoimax(
  const char * restrict str,
  char ** restrict ptr,
  int base);
uintmax_t strtoumax(
  const char * restrict str,
  char ** restrict ptr,
  int base);
```

这些函数把字符串转换为最大长度的整数，它们所采取的方式与stdlib.h所定义的strtol和stroul函数相同。如果结果将导致溢出，则返回INTMAX_MAX、INTMAX_MIN或UINTMAX_MAX中合适的一个，并且把errno设置为ERANGE。

参考：errno和ERANGE 第11.2节；strtol和strtoul 第16.4节。

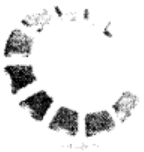
21.9 wcstoimax、wcstoumax

语法概要

```
#include <stddef.h>     // wchar_t
#include <inttypes.h>
intmax_t wcstoimax(
  const wchar_t * restrict str,
  wchar_t ** restrict ptr,
  int base);
uintmax_t wcstoumax(
  const wchar_t * restrict str,
  wchar_t ** restrict ptr,
  int base);
```

这些函数把宽字符串转换为最大长度的整数，它们所采取的方式与wchar.h所定义的wcstol和wcstoul函数相同。如果结果将导致溢出，则返回INTMAX_MAX、INTMAX_MIN或UINTMAX_MAX中合适的一个，并且把errno设置为ERANGE。

参考：errno和ERANGE 第11.2节；wcstol和wcstoul 第24章。



第22章
浮点环境

本节所讨论的工具是C99新增的，它们对float.h中的信息进行了补充。它们向那些需要对浮点运算的精度或性能进行高度控制的应用程序提供了访问浮点环境的能力。这些工具是在头文件fenv.h中定义的。

参考：float.h 表5-3。

22.1 介　绍

编写高精度浮点算法的程序员需要控制浮点环境的各个方面的特性：如何对结果进行四舍五入、如何对浮点表达式进行简化或转换、像下溢这样的浮点事件应该被忽略还是应该看成是程序错误。对浮点环境的控制是通过设置*控制模式*（control mode）进行的，它影响浮点运算是如何实现的。浮点运算通过产生浮点异常向程序员提供反馈，这种异常可能会中断C程序的控制流，并记录在程序员可以读取的*状态标志*（status flag）中。C99程序员还可以通过使用第17章所列出的专用的浮点数学函数控制浮点运算的行为。

浮点运算可能在两个不同的时间执行。当编译器编译C程序时，将会执行常量（编译时）浮点运算；当C程序动态运行（执行时）时，也可能执行浮点运算。C99标准只对运行时的运算提供了明确的控制。编译器可以提供自己的工具，对编译时的运算进行控制。

C99所引用的国际浮点标准是IEC 60559:989《微处理器系统的二进制浮点运算，第2版》。这个标准的前身是IEC 559:1989和ANSI/IEEE 754-1985《二进制浮点运算的IEEE标准》（IEEE 754标准后来进行了通用化，根据ANSI/IEEE 854-1987《基数独立的浮点运算IEEE标准》去除了对基数和字长的依赖）。C99标准的附录F详细说明了C语言到IEC 60559的映射。后者是可选的，除非C编译器显式定义了__STDC__IEC_559__宏。

程序约定

控制浮点行为的工具是动态的。也就是说，当程序执行期间使用本章的函数作出一个改变时，这个改变就会一直有效，直到下一次显式地作出改变。一个特定的函数是如何执行浮点操作的取决于最近被调用的是fenv.h中的哪些函数，因此无法在C程序被编译时确定。当底层的硬件使用全局控制寄存器控制浮点运算时，不会存在什么问题，如果实际的操作码是由控制这个行为的编译器发出的，就比较难以实现。

C99标准推荐程序员总是假设所有被调用的函数将采用默认的浮点行为，除非文档另有陈述。类似，被调用的函数不应该更改环境，除非文档明确表示需要这样做。也就是说，当一个函数被调用时，它不应该依赖于任何状态标志，也不应该更改已经生效的标志。如果必要，它可以期望默认的控制模式生效，并且不应该修改调用者的模式。任何函数都可能会产生浮点异常。

22.2 浮点环境

语法概要

```
#include <fenv.h>
#pragma STDC FENV_ACCESS   on-off-switch
typedef ... fenv_t;
#define FE_DEFL_ENV ...
int fegetenv(fenv_t *envp);
int fesetenv(fenv_t *envp);
int feholdexcept(fenv_t *envp);
int feupdateenv(const fenv_t *envp);
```

标准pragma指令FENV_ACCESS表示C程序是否将设置浮点控制模式、测试状态标志、是否在非默认控制模式下也将运行。当FENV_ACCESS被“关闭”时，这些操作的行为是未定义的。提供这条pragma指令是为了防止这些信息对C程序的编译或优化产生明显的差别。默认设置是由编译器所定义的，因此关心移植性问题的程序员应该总是认定它是“关闭”的。pragma指令FENV_ACCESS遵循标准pragma的正常替换规则。

fenv_t类型是编译器所定义的，用于保存完整的浮点状态，包括控制模式和异常状态位。

FE_DEFL_ENV宏展开之后，把默认的浮点环境指定为一个fenv_t *类型的值。C编译器可能会定义其他以FE_和一个大写字母开头的宏。程序员应该把这些宏看成是指定了只读对象。

fegetenv函数提取当前浮点环境，并把它存储在envp所指向的对象中。如果调用成功，它返回0，否则返回一个非零值。

fesetenv函数用envp所指向的环境替换当前浮点环境。这个环境必须是以前由fegetenv或feholdexcept设置的，或者是个像FE_DEFL_ENV这样的预定义环境。如果调用成功，它返回0，否则返回一个非零值。

feholdexcept函数一般用于关闭一段时间的浮点异常。这个函数在envp所指向的对象中保存当前浮点环境，然后安装一个忽略所有浮点异常的环境。如果这个“非停顿”的环境安装成功，这个函数返回0，否则返回一个非零值。有些编译器可能无法忽略所有的异常。

feupdateenv函数在一些局部存储区域中保存当前所引发的浮点异常，恢复envp所指向的环境为新环境，并最终引发它所保存的异常。如果调用成功，它返回0，否则返回一个非零值。

参考：pragma和替换规则 第3.7节；引发浮点异常 第22.3节。

22.3 浮点异常

语法概要

```
#include <fenv.h>
macro FE_DIVBYZERO ...
macro FE_INEXACT ...
macro FE_INVALID ...
macro FE_OVERFLOW ...
macro FE_UNDERFLOW ...
...
macro FE_ALL_EXCEPT ...
typedef ... fexcept_t;
```

（续）

语法概要

```
int fegetexceptflag(fexcept_t *flagp, int excepts);
int fesetexceptflag(const fexcept_t *flagp, int excepts);
int fetestexcept(int excepts);
int feraiseexcept(int excepts);
int feclearexcept(int excepts);
```

浮点异常是某些浮点运算的副作用。所有的异常都会设置一个状态标志，提示发生了指定的异常。至于这个异常是否会中断程序的控制流，则取决于浮点控制模式的设置。

fexecpt_t类型是编译器所定义的，用于保存编译器所支持的所有浮点状态标志。它通常是个整数类型，不同的位分别表示不同的异常，但也可以包含其他更详尽的信息。例如，fexecpt_t可以包含状态标志是在什么地方被触发的信息。

C编译器可以支持不同的浮点异常。对于每种受支持的异常，编译器必须定义一个像FE_DIVBYZERO、FE_INEXACT、FE_INVALID、FE_OVERFLOW和FE_UNDERFLOW这样的宏。未支持的异常必须不予定义（例如，不是把它定义为零）。每个被定义的宏的展开结果是一个整数常量表达式，而且这些值必须可以通过位或的方式进行组合，表示一个任意的异常子集。一般情况下，每个宏被展开为2的一个不同整数次方。FE_ALL_EXCEPT宏是所有受支持的异常的位或组合。它是根据本节中的函数的签名所确定的，并且它所表示的异常的数量不能超过int类型的位数（int类型至少为16位）。

fegetexceptflag函数把当前的浮点状态标志的设置存储在flagp所指向的对象中。并不是所有的状态标志都被存储到flagp *中，只有excepts参数所列出的那些异常才会被保存。flagp *中的其他状态标志保持不变。excepts参数作为“受关注”的异常的掩码。如果调用成功，这个函数返回0，否则返回一个非零值。

fesetexceptflag函数把当前的浮点状态标志设置为flagp所指向的对象所保存的值。注意，并不是所有的状态标志都被设置，只有excepts参数所列出的那些异常才会被设置，其他状态标志保持不变。excepts参数作为“受关注”的异常的掩码。如果所有受关注的标志都被设置为适当的状态，这个函数返回0，否则返回一个非零值。

fetestexcept函数返回与异常标志对应的异常宏的位或测试结果，它们通常是excepts参数中所出现的环境变量的当前设置。因此，fetestexcept返回excepts中当前所设置的异常的一个子集。

feraiseexcept函数引发excepts参数所表示的异常。这些异常的引发顺序并未指定，并且可能有一些异常由于副作用而引发其他的异常。例如，FE_INEXACT常常与其他异常一起发生。

feclearexcept函数清除与excepts参数对应的异常的当前异常状态标志。如果excepts中的所有异常都被清除，这个函数返回0，否则返回一个非零值。

22.4 浮点四舍五入模式

语法概要

```
#include <fenv.h>
macro FE_DOWNWARD ...
macro FE_UPWARD ...
```

（续）

语法概要

```
macro FE_TONEAREST ...
macro FE_TOWARDZERO ...
int fegetround(void);
int fetestround(int rounds);
```

C99编译器必须定义像FE_DOWNWARD、FE_UPWARD、FE_TONEAREST或FE_TOWARDZERO这样的宏，它们表示四舍五入的方向，可以由本节所讨论的函数所获取和设置。这些宏展开后的值是可以用int类型表示的不同的非负整数常量。未受支持的舍入方向将不会定义对应的宏。

fegetround函数返回当前的四舍五入方向，这个方向是用其中一个四舍五入方向宏表示的。类似，fesetround函数设置当前的四舍五入方向，如果调用成功就返回0。如果无法获取或设置四舍五入的方向，这两个函数都返回一个负值。

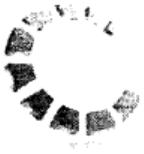
22.5 浮点表达式收缩

语法概要

```
#pragma STDC FP_CONTRACT on-off-switch
```

C99允许利用组合了多个操作符的快速计算机指令对浮点表达式进行优化（或收缩）。有些指令（例如常见于数字信号处理器的乘-加指令）在四舍五入或其他行为上可能与C抽象机的要求不同。

收缩是由标准pragma指令FP_CONTRACT控制的，如果开关值为ON，就允许收缩。如果开关值为OFF，收缩就被禁止。这条prgama指令的默认设置取决于编译器，并且遵循标准prgama指令的正常替换规则（参见第3.7.2节）。



第23章 复数运算

本章所讨论的工具支持复数运算。它们是在C99的头文件complex.h中定义的。

23.1 复数函数库约定

所有的角度都是用弧度表示的。复数z可以写为x + yi，其中x和y都是实数。类似，w = u + vi和c = a + bi也是复数。

对于具有分支并且函数在分支处断开的复数函数，应该采用下面这些由编译器所定义的约定之一。如果编译器采用了有符号的0，则0的符号可以用于区分分支的两侧。否则，当函数从逆时钟方向趋向分支的有穷端时，库函数实现应该把复数函数看成是连续的。

参考：复数类型 第5.2.1节。

23.2 complex、_Complex_I、imaginary、_Imaginary_I、I

语法概要

```
#include <complex.h>                    // All C99
#define complex _Complex
#define imaginary _Imaginary
#define _Complex_I    ...
#define _Imaginary_I  ...
#define I ...
```

如果编译器支持复数类型，则complex宏就被定义为关键字_Complex的同义词。如果编译器支持虚数类型，则imaginary宏就被定义为关键字_Imaginary的同义词。如果各自的类型均得到支持，则_Complex_I和_Imaginary_I分别被定义为const float _Complex和const float _Imaginary类型的常量表达式，它们的值是虚数单位 $\sqrt{(-1)}$ 或i。

如果编译器支持复数类型，则I宏展开为_Complex_I。如果已经定义了虚数类型，则I也可能被展开为_Imaginary_I。

由于C99之前所编写的程序可能使用了标识符complex、imaginary和I，因此允许#undifine或重新定义这些宏。

参考：复数类型 第5.2.1节。

23.3 CX_LIMITED_RANGE

语法概要

```
#include <complex.h>                    // All C99
#pragma STDC CX_LIMITED_RANGE on-or-off-switch
```

标准pragma指令CX_LIMITED_RANGE如果被“打开”，它通知编译器使用复数乘、除和绝

对值运算的“明显”实现是可以接受的。这条pragma指令的默认状态是“关闭”。pragma指令CX_LIMITED_RANGE遵循标准pragma指令的替换规则。“明显”实现包括：

乘法：$z * w = (x + iy)(u + iv) = (xu - yv) + i(yu + xv)$

除法：$z/w = (x + iy) / (u + iv) = ((xu + yv) + i(yu - xv)) / (u^2 + v^2)$

绝对值：$|z| = |x + iy| = \sqrt{(x^2 + y^2)}$

这些实现都是“数值挑战的”，因为它们存在潜在的不必要的下溢和上溢，并且它们无法处理无限。但是，只要程序员知道它们在当前的程序中是安全的，它们的速度可能非常快。

参考：标准pragma、“打开”－“关闭”开关和替换规则 第3.7节。

23.4 cacos、casin、catan、ccos、csin、ctan

语法概要

```
#include <complex.h>                          // All C99

double complex cacos (double complex z);
float complex cacosf(float complex z);
long double complex cacosl(long double complex z);

double complex casin (double complex z);
float complex casinf(float complex z);
long double complex casinl(long double complex z);

double complex catan (double complex z);
float complex catanf(float complex z);
long double complex catanl(long double complex z);

double complex ccos (double      complex z);
float complex ccosf(float complex z);
long double complex caosl(long double complex z);

double complex csin (double      complex z);
float complex csinf(float complex z);
long double complex csinl(long double complex z);

double complex ctan (double      complex z);
float complex ctanf(float complex z);
long double complex ctanl(long double complex z);
```

表23-1列出了这些函数的定义域和值域，假设具有下面的定义

$$(a + bi) = f(x + yi)$$

表23-1 复数三解函数的定义域和值域

C名称	函数	分支	值的范围
cacos	复数反余弦	$y = 0, x > +1$和$y = 0, x < -1$	$0 \leqslant a \leqslant \pi$
casin	复数反正弦	$y = 0, x > +1$和$y = 0, x < -1$	$-\pi/2 \leqslant a \leqslant +\pi/2$
catan	复数反正切	$x = 0, y > +1$和$x = 0, y < -1$	$-\pi/2 \leqslant a \leqslant +\pi/2$
ccos	复数余弦		
csin	复数正弦		
ctan	复数正切		

23.5 cacosh、casinh、catanh、ccosh、csinh、ctanh

语法概要

```
#include <complex.h>                          // All C99

double complex cacosh (double complex z);
float complex cacoshf(float complex z);
long double complex cacoshl(long double complex z);

double complex casinh (double complex z);
float complex casinhf(float complex z);
long double complex casinhl(long double complex z);

double complex catanh (double complex z);
float complex catanhf(float complex z);
long double complex catanhl(long double complex z);

double complex ccosh (double complex z);
float complex ccoshf(float complex z);
long double complex ccoshl(long double complex z);

double complex csinh (double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);

double complex ctanh (double complex z);
float complex ctanhf(float complex z);
long double complex ctanhl(long double complex z);
```

表23-2列出了这些函数的定义域和值域，假设：

$$(a + bi) = f(x + yi)$$

表23-2 复数双曲函数的定义域和值班

C名称	函数	分支	值的范围
cacosh	复数反双曲余弦	$y = 0, x < +1$	$0 \leqslant a, -\pi \leqslant b \leqslant +\pi$
casinh	复数反双曲正弦	$x = 0, y > +1$和$x = 0, y < -1$	$-\pi/2 \leqslant b \leqslant +\pi/2$
catanh	复数反双曲正切	$y = 0, x > +1$和$y = 0, x < -1$	$-\pi/2 \leqslant b \leqslant +\pi/2$
ccosh	复数双曲余弦		
csinh	复数双曲正弦		
ctanh	复数双曲正切		

23.6 cexp、clog、cabs、cpow、csqrt

语法概要

```
#include <complex.h>                          // All C99

double      complex cexp (double complex z);
float       complex cexpf(float       complex z);
long double complex cexpl(long double complex z);

double      complex clog (double complex z);
float       complex clogf(float       complex z);
long double complex clogl(long double complex z);

double              cabs (double complex z);
```

（续）

语法概要

```
float                cabsf(float       complex z);
long double cabsl(long double complex z);

double      complex cpow (
    double complex z,
    double complex u);
float       complex cpowf(
    float complex z,
    float complex u);
long double complex cpowl(
    long double complex z,
    long double complex u);

double      complex csqrt (double complex z);
float       complex csqrtf(float complex z);
long double complex csqrtl(long double complex z);
```

表23-3列出了这些函数的定义域和值域，假设具有下面的定义

$$(a + bi) = f(z) = f(x + yi) \text{ 或}$$
$$(a + bi) = f(z, w) = f(x + yi, u + vi)$$

表23-3 复数指数和幂函数的定义域和值域

C名称	函数	分支	值的范围
cexp	e^z		
clog	$\ln z$	$y = 0, x < 0$	$-\pi \leqslant b \leqslant +\pi$
cabs	绝对值 $a = \text{sqrt}(x^2 + y^2)$		
cpow	z^w	$y = 0, x < 0$	
csqrt	平方根	$y = 0, x < 0$	

23.7 carg、cimag、creal、conj、cproj

语法概要

```
#include <complex.h>                        // All C99

double      carg (double complex z);
float       cargf(float       complex z);
long double cargl(long double complex z);

double      cimag (double complex z);
float       cimagf(float       complex z);
long double cimagl(long double complex z);

double      creal (double complex z);
float       crealf(float       complex z);
long double creall(long double complex z);

double complex conj (double      complex z);
float complex conjf(float complex z);
long double complex conjl(long double complex z);

double complex cproj (double complex z);
float complex cprojf(float complex z);
long double complex cprojl(long double complex z);
```

表23-4列出了这些函数的定义域和值域，假设具有下面的定义

$$(a + bi) = f(x + yi) \text{ 或}$$
$$(a + bi) = f(x + yi, u + vi)$$

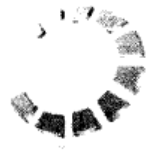
表23-4 各种复数函数的定义域和值域

C名称	函数	分支	值的范围
carg	参数（又称相位角）	$y = 0, x < 0$	$[-\pi, +\pi]$
cimag	z的虚部：y		
creal	z的实部：x		
conj	$a = x, b = -y$		
cproj	Riemann球体上的投影		

carg(z)的值是复数平面上从正的实数轴到从原点到z的直线之间的角度。

如果z不是无限，cproj(z)的值就是z。如果z是无限，则cproj(z)是用一个复数表示的正的实数无限。如果编译器支持有符号的零并且z是无限的，则cproj(z)的（零）虚部将和z的虚部具有相同的符号。为了使z是无限的，只要z的任一成员是无限的就可以了，即使其他两个成员都是NaN。

第24章



宽字符和多字节字符工具

本章所讨论的工具支持宽字符和宽字符串，以及多字节字符和多字节字符串。wctype.h头文件提供了字符分类和映射工具，其余的字符和字符串工具则位于wchar.h。本章所讨论的大部分工具复制了ctype.h、string.h和stdio.h中的传统字符和字符串工具的功能，只是更改了它们的参数和返回值的类型。

24.1 基本类型和宏

语法概要

```
#include <wchar.h>
typedef ... wchar_t;
typedef ... wint_t;
typedef ... mbstate_t;
typedef ... size_t;

#define WEOF ...
#define WCHAR_MIN ...
#define WCHAR_MAX ...
```

wchar_t类型（宽字符类型）是一种整数类型，它可以表示受支持的区域中的任何执行时扩展字符集中的所有不同字符。它可以是有符号或无符号类型，并且还在stddef.h头文件中进行了定义。WCHAR_MIN和WCHAR_MAX宏提供了wchar_t类型的数值限制。它们的值并不一定要对应于扩展字符。

wint_t类型也是一种整数类型，它除了可以表示所有的wchar_t类型的值之外，至少还可以表示一个不属于扩展字符集内容的额外值。这个常量是由WEOF宏给出的，表示“输入的结束”以及其他异常条件。wint_t类型在寻常参数提升中并不会被改变。

mbstate_t是一种非数组对象类型，可以表示多字节字符序列和宽字符串之间的一种转换状态。

size_t类型与stddef.h头文件所定义的同名类型相同。

参考：size_t 第11.1节；wchar_t 第11.1节；宽字符 第2.7.3节。

24.2 宽字符和多字节字符之间的转换

语法概要

```
#include <wchar.h>
size_t mbrlen(const char *s, size_t n, mbstate_t *ps);
wint_t btowc(int c);
size_t mbrtowc(wchar_t *pwc, const char *s, size_t n,
    mbstate_t *ps);
int wctob(wint_t c);
size_t wcrtomb(char *s, wchar_t wc, mbstate_t *ps);
int mbsinit(const mbstate_t *ps);
```

本节的转换函数是stdlib.h中的基本函数：mbtowc和wctomb（第16.10节）的扩展版本。这些函数是在C89修正案1中添加的，它们具有更强的灵活性，并且它们的行为得到了更完整的定义。

mbrlen函数检查s所指定的字符串的前n个字节，观察这些字符是否表示一个相对于ps所保存的转换状态而言是合法的多字节字符。如果ps为null，则这个函数使用它自己的内部状态（它是在程序启动时初始化为初始状态的）。如果s是个null指针，这个调用就把s看成是""并把n看成是1。如果s是合法的，并且对应于null宽字符，则这个函数返回0（不管有多少个字符组成了这个多字节字符）。如果s是其他任何合法的多字节字符，则这个函数返回组成这个字符的字节数量（即返回值的范围位于1到n之间）。如果s是个不完整的多字节字符，则这个函数返回-2。如果s是个非法的多字节字符，则这个函数返回-1，并且errno被设置为EILSEQ。当返回值非负时，转换状态被更新。如果返回值为-1，转换状态是未定义的。如果返回值为-2，则转换状态不会发生变化。

btowc函数返回对应于字节c的宽字符，c被看成是个单字节的处于初始转换状态的多字节字符。如果c（转换为unsigned char）并不对应于一个合法的多字节字符，或者如果c是EOF，则btowc返回WEOF。

mbrtowc函数根据转换状态ps把一个多字节字符s转换为一个宽字符（如果ps为null，则使用一个内部状态对象，它是在程序启动时设置为初始状态的）。如果pwc不是个null指针，则转换结果存储在pwc所指定的对象中。如果s是个null指针，则调用mbrtowc相当于调用mbrtowc(NULL, "", 1, ps)。也就是说，s被看成是个空字符串，pwc和n的值被忽略。如果s是个对应于null宽字符的合法字符，则这个函数返回0（不管使用了多少个字节）。否则，如果s是个合法的多字节字符，则这个函数返回被使用的字节数量。如果s是个不完整的多字节字符，则这个函数返回-2。最后，如果s是个非法的多字节字符，则这个函数返回-1。当执行了合法的转换时，ps所指定的转换状态（如果ps为null指针，则是内部转换状态）就被更新。如果s是不完整的，则转换状态保持不变。如果s是非法的，则转换状态是未定义的。

wctob函数（C89修正案1）返回处于初始转换状态的与宽字符c对应的单字节的多字节字符。如果不存在这样的单字节，则返回EOF。

wcrtomb函数把一个宽字符wc转换为一个相对于ps所指定的转换状态的多字节字符（如果ps为null，则使用一个内部转换状态对象）。这个多字节字符被存储在一个数组中，这个数组的第一个元素是由s所指定的，它的长度至少必须是MB_CUR_MAX个字符。转换状态将被更新。如果wc是个null宽字符，则存储一个null字节，并在前面添加恢复为初始转换状态所需要的任何转移序列。这个函数返回存储在s中的字符数量。如果s是个null指针，则wc被忽略，调用wcrtomb的效果就是简单地恢复为初始转换状态并返回1（就像L'\0'被转换为一个隐藏缓冲区一样）。如果wc并不是一个合法的宽字符，则这个函数把errno设置为EILSEQ，并返回-1。

如果ps是个null指针或是个指向一个表示初始转换状态的对象的指针，mbsinit函数返回一个非零值，否则就返回0。

参考：EILSEQ 第11.2节；errno 第11.2节；多字节字符 第2.1.5节；mbstate_t 第11.1节；size_t 第11.1节；wchar_t 第11.1节；wint_t 第11.1节。

24.3 宽字符串和多字节字符串之间的转换

语法概要

```
#include <wchar.h>
size_t mbsrtowcs(wchar_t *pwcs, const char **src, size_t n,
    mbstate_t *ps);
size_t wcsrtombs(char *s, const wchar_t **src, size_t n,
    mbstate_t *ps);
```

本节所讨论的函数是stdlib.h（参见第16.11节）所定义的mbstowcs和wcstombs函数的“可重新开始”版本。这些函数是在C89修正案1中添加的。

mbsrtowcs函数把一个以null字符结尾的字符串s中的一个多字节字符序列转换为一个对应的宽字符序列，并把结果存储在pwcs所指定的数组中。初始转换状态是由ps所指定的，多字节字符的输入序列是间接地由src所指定的。在正常的操作中，每个多字节字符（直到结尾的null字符，且包括该null字符）就像调用了mbrtowc一样被转换，输出的宽字符被放置在pwcs所指定的字符数组中。在转换之后，src所指定的指针被设置为null指针，表示输入字符串已经被转换。这个函数返回pwcs中所存储的宽字符数量（不包括结尾的null宽字符）。转换状态被更新为初始转换状态，这是对输入的多字节字符串尾部的null字符进行转换的结果。输出指针pwcs可以是null指针。在这种情况下mbsrtowcs简单地计算转换所需要的输出宽字符串的长度。

如果发生了转换错误，输入多字节字符串的转换将会永久终止。在这种情况下，src所指定的指针被更新为指向试图进行转换时发生错误的那个多字节字符。此时，这个函数返回−1，并把errno设置为EILSEQ，并且转换状态将是不确定的。

wcsrtombs函数把一个以pwcs指定的值开始的宽字符序列转换为一个多字节字符序列，把结果存储在s所指定的字符数组中。初始转换状态是由ps所指定的，输入的宽字符串是间接地由src所指定的。在正常的操作中，每个宽字符（直到结尾的null字符，且包括该null宽字符）就像调用了wcrtomb一样被转换，输出的多字节字符被放置在s所指定的字符数组中。在转换之后，src所指定的指针被设置为null指针，表示整个输入字符串已经被转换，并且这个函数返回s中所存储的多字节字符的数量（不包括结尾的null字符）。转换状态将被更新为初始转换状态，这是对输入的宽字符串尾部的null字符进行转换的结果。输出指针s可以是null指针，在这种情况下wcsrtombs简单地计算转换所需要的输出字符数组的长度。

如果已经有n个输出字节被写入到s（并且s不是null指针），输入宽字符串的转换将会在结尾的null宽字符被转换之前终止。在这种情况下，src所指定的指针被设置为指向最后一个被转换的宽字符之后的位置。转换状态将被更新，它并不一定是初始状态。此时，这个函数返回n。

如果发生了转换错误，输入宽字符串的转换将会永久终止。在这种情况下，src所指定的指针被更新为指向试图进行转换时发生错误的那个宽字符。此时，这个函数返回−1，并把errno设置为EILSEQ，并且转换状态将是不确定的。

参考：转换状态 第2.1.5节；多字节字符 第2.1.5节；宽字符 第2.1.5节。

24.4 转换为算术类型

语法概要

```
#include <wchar.h>

double wcstod(
  const wchar_t * restrict str,
  wchar_t ** restrict ptr );
float wcstof(
  const wchar_t * restrict str,
  wchar_t ** restrict ptr );
long double wcstold(
  const wchar_t * restrict str,
  wchar_t ** restrict ptr );

long wcstol(
  const wchar_t * restrict str,
  wchar_t ** restrict ptr, int base );
long long wcstoll(
  const wchar_t * restrict str,
  wchar_t ** restrict ptr, int base );
unsigned long wcstoul(
  const char * restrict str,
  wchar_t ** restrict ptr, int base );
unsigned long strtoull(
  const char * restrict str,
  wchar_t ** restrict ptr, int base );
```

本节中的wcsto...函数与第16.4节中对应的strto...函数相同，区别在于参数的类型，并且这些函数使用iswspace函数检查空格，并使用小数点宽字符代替点号。除了strto...函数所接受的字符串之外，这些宽字符转换函数还可以接受编译器所定义的输入字符串。

wcstod、wcstol和wcstoul函数是C89修正案1所增加的，其余的函数是C99所增加的。

24.5 输入和输出函数

表24-1列出了用于输入和输出宽字符串的函数，并列出了它们对应的字节版本，以及它们在本书中所出现的章节。

表24-1 宽字符输入/输出函数

宽字符函数	所在章节	字节字符函数
fgetwc	15.6	fgetc
fgetws	15.7	fgets
fputwc	15.9	fputc
fputws	15.10	fputs
fwide	15.2	
fwprintf	15.11	fprintf
fwscanf	15.8	fscanf
getwc	15.6	getc
getwchar	15.6	getchar
putwc	15.9	putc

（续）

宽字符函数	所在章节	字节字符函数
putwchar	15.9	putchar
swprintf	15.11	sprintf
swcanf	15.8	sscanf
ungetwc	15.6	ungetc
vfwprintf	15.12	vfprintf
vfwscanf	15.12	vfscanf
vswprintf	15.12	vsprintf
vswscanf	15.12	vsscanf
vwprintf	15.12	vprintf
vwscanf	15.8	vscanf
wprintf	15.11	printf
wscanf	15.8	scanf

24.6 字符串函数

表24-2列出了支持宽字符串的函数，并列出了它们对应的字节版本，以及它们在本书中所出现的章节。

表24-2 宽字符串函数

宽字符函数	所在章节	字节字符函数
wcscat	13.1	strcat
wcschr	13.5	strchr
wcscmp	13.2	strcmp
wcscoll	13.10	strcoll
wcscpy	13.3	strcpy
wcscspn	13.6	strcspn
wcslen	13.4	strlen
wcsncat	13.1	strncat
wcsncmp	13.2	strncmp
wcsncpy	13.3	strncpy
wcspbrk	13.6	strpbrk
wcsrchr	13.5	strrchr
wcsspn	13.6	strspn
wcsstr	13.7	strstr
wcstok	13.7	strtok
wcsxfrm	13.10	strxfrm
wmemchr	14.1	memchr
wmemcmp	14.1	memcmp
wmemcpy	14.3	memcpy
wmemmove	14.3	memmove
wmemset	14.4	memset

24.7 日期和时间转换

wcsftime宽字符函数对应于strftime字节函数。

参考：strftime 第18.6节。

24.8 宽字符分类和映射函数

表24-3列出了宽字符分类和映射函数，并列出了与它们对应的字符函数，以及它们在本书中所出现的章节。

宽字符函数towctrans不存在平行的函数。它的签名是：

```
#include <wctype.h>
wint_t towctrans( wint_t wc, wctrans_t desc );
```

towctrans函数把宽字符wc映射为一个新值，并返回这个值。映射是由一个wctrans_t类型的值所指定的，这个值可以通过调用wctrans函数（第12.11节）来获取。在towctrans调用期间，LC_CTYPE区域分类设置必须相同，就像调用wctrans_t期间一样，它产生desc的值。

表24-3 宽字符函数

宽字符函数	章节	字节字符函数
iswalnum	12.1	isalnum
iswalpha	12.1	isalpha
iswblank	12.	isblank
iswcntrl	12.1	iscntrl
iswctype	12.	isctype
iswdigit	12.3	isdigit
iswgraph	12.4	isgraph
iswlower	12.5	islower
iswprint	12.4	isprint
iswpuct	12.4	ispunct
iswspace	12.6	isspace
iswupper	12.5	isupper
iswxdigit	12.3	isxdigit
towlower	12.9	tolower
towupper	12.9	toupper
wctrans	12.11	ctrans

附录A
ASCII字符集

十六进制	八进制	0 0 十进制	字符	名称	0x20 040 十进制	字符	0x40 0100 十进制	字符	0x60 0140 十进制	字符
0	0	0	^@	NUL	32	SP	64	@	96	`
1	1	1	^A	SOH	33	!	65	A	97	a
2	2	2	^B	STX	34	"	66	B	98	b
3	3	3	^C	ETX	35	#	67	C	99	c
4	4	4	^D	EOT	36	$	68	D	100	d
5	5	5	^E	ENQ	37	%	69	E	101	e
6	6	6	^F	ACK	38	&	70	F	102	f
7	7	7	^G	BEL, \a	39	'	71	G	103	g
8	010	8	^H	BS, \b	40	(	72	H	104	h
9	011	9	^I	TAB, \t	41	)	73	I	105	i
0xA	012	10	^J	LF, \n	42	*	74	J	106	j
0xB	013	11	^K	VT, \v	43	+	75	K	107	k
0xC	014	12	^L	FF, \f	44	,	76	L	108	l
0xD	015	13	^M	CR, \r	45	-	77	M	109	m
0xE	016	14	^N	SO	46	.	78	N	110	n
0xF	017	15	^O	SI	47	/	79	O	111	o
0x10	020	16	^P	DLE	48	0	80	P	112	p
0x11	021	17	^Q	DC1	49	1	81	Q	113	q
0x12	022	18	^R	DC2	50	2	82	R	114	r
0x13	023	19	^S	DC3	51	3	83	S	115	s
0x14	024	20	^T	DC4	52	4	84	T	116	t
0x15	025	21	^U	NAK	53	5	85	U	117	u
0x16	026	22	^V	SYN	54	6	86	V	118	v
0x17	027	23	^W	ETB	55	7	87	W	119	w
0x18	030	24	^X	CAN	56	8	88	X	120	x
0x19	031	25	^Y	EM	57	9	89	Y	121	y
0x1A	032	26	^Z	SUB	58	:	90	Z	122	z
0x1B	033	27	^[	ESC	59	;	91	[	123	{
0x1C	034	28	^\	FS	60	<	92	\	124	\|
0x1D	035	29	^]	GS	61	=	93	]	125	}
0x1E	036	30	^^	RS	62	>	94	^	126	~
0x1F	037	31	^_	US	63	?	95	_	127	DEL



附录B
语　法

抽象声明器：
　　指针
　　指针$_{opt}$　直接抽象声明器
加减表达式：
　　乘除表达式
　　加减表达式　加减操作符　乘除表达式
加减操作符：下列之一
　　+　-
地址表达式：
　　& 类型转换表达式
数组声明器：
　　直接声明器 [常量表达式]　　(直到C99)
　　直接声明器 [数组限定符列表$_{opt}$ 数组长度表达式$_{opt}$]　　(C99)
　　直接声明器 [数组限定符列表$_{opt}$ *]　　(C99)
数组限定符：
　　static
　　restrict
　　const
　　volatile
数组限定符列表：
　　数组限定符
　　数组限定符列表　数组限定符
数组长度表达式：
　　赋值表达式
　　*
赋值表达式：
　　条件表达式
　　单目表达式　赋值操作符　赋值表达式
赋值操作符：下列之一
　　=　+=　-=　*=　/=　%=　<<=　>>=　&=　^=　|=
二进制指数：
　　p　符号部分$_{opt}$　数字序列
　　p　符号部分$_{opt}$　数字序列
位段：
　　声明器$_{opt}$：宽度
位与表达式：
　　相等表达式
　　位与表达式 & 相等表达式
位反表达式：
　　~ 类型转换表达式
位或表达式：

位异或表达式
位或表达式 | 位异或表达式

位异或表达式：
位与表达式
位异或表达式 ^ 位与表达式

break语句：
break ;

case标签：
case 常量表达式

类型转换表达式：
单目表达式
(类型名) 类型转换表达式

c字符：
除了单引号（'）、反斜杠（\）或换行符之外的其他任何源字符
转义字符
统一字符名称 (C99)

c字符序列：
c字符
c字符序列 c字符

字符常量：
'c字符序列' (C89)
L'c字符序列'

字符转义码：下列之一
n t b r f
v \ ' "
a ? (C89)

字符类型指定符：
char
signed char
unsigned char

逗号表达式：
赋值表达式
逗号表达式， 赋值表达式

复数类型指定符： (C99)
float _Complex
double _Complex
long double _Complex

成员声明：
类型指定符 成员声明器列表

成员声明器：
简单成员
位段

成员声明器列表：
成员声明器
成员声明器列表，成员声明器

成员选择表达式：
直接成员选择



间接成员选择

复合字面值： (C99)

(类型名) { 初始化值列表 ,$_{opt}$ }

复合语句：

{ 声明或语句列表$_{opt}$ }

条件表达式：

逻辑或表达式

逻辑或表达式 ? 表达式 : 条件表达式

条件语句：

if 语句

if - else 语句

常量：

整数常量

浮点常量

字符常量

字符串常量

常量表达式：

条件表达式

continue语句：

continue ;

十进制常量：

非零数字

十进制常量数字

二进制浮点常量：

数字序列 指数 浮点后缀$_{opt}$

带点号的数字 指数$_{opt}$ 浮点后缀$_{opt}$

声明：

声明指定符 初始化声明器列表

声明列表：

声明

声明列表 声明

声明或语句：

声明

语句

声明或语句列表：

声明或语句

声明或语句列表 声明或语句

声明指定符：

存储类别指定符 声明指定符$_{opt}$

类型指定符 声明指定符$_{opt}$

类型限定符 声明指定符$_{opt}$

函数指定符 声明指定符$_{opt}$ (C99)

声明器：

指针声明器

直接声明器

default标签：

default

指定：

指定符列表 =

指定符：

[常量表达式]

.标识符

指定符列表：

指定符

指定符列表 指定符

数字：下列之一

0 1 2 3 4 5 6 7 8 9

数字序列：

数字

数字序列 数字

直接抽象声明器：

(抽象声明器)

直接抽象声明器$_{opt}$ [常量表达式$_{opt}$]

直接抽象声明器$_{opt}$ [表达式] (C99)

直接抽象声明器$_{opt}$ [*] (C99)

直接抽象声明器$_{opt}$ [参数类型列表$_{opt}$]

直接成员选择：

后缀表达式. 标识符

直接声明器：

简单声明器

(声明器)

函数声明器

数组声明器

do语句：

do 语句 while (表达式) ;

带点号的数字：

数字序列 .

数字序列 . 数字序列

. 数字序列

带点号的十六进制数字：

十六进制数字序列 .

十六进制数字序列 . 十六进制数字序列

. 十六进制数字序列

枚举常量：

标识符

枚举常量定义：

枚举常量

枚举常量 = 表达式

枚举定义列表：

枚举常量定义

枚举定义列表，枚举常量定义

枚举标签：

标识符

枚举类型定义：

enum 枚举标签$_{opt}$ {枚举定义列表}
enum 枚举标签$_{opt}$ {枚举定义列表，} (C99)

枚举类型引用：
enum s 枚举标签

枚举类型指定符：
枚举类型定义
枚举类型引用

相等表达式：
关系表达式
相等表达式 相等操作符 关系表达式

相等操作符：下列之一
== !=

转义字符：
\ 转义码 (C99)
统一字符名称

转义码：
字符转义码
八进制转义码
十六进制转义码 (C89)

指数：
e 符号部分$_{opt}$ 数字序列
E 符号部分$_{opt}$ 数字序列

表达式：
逗号表达式

表达式列表：
赋值表达式
表达式列表，赋值表达式

表达式语句：
表达式 ;

字段列表：
成员声明
字段列表 成员声明

浮点常量：
十进制浮点常量
十六进制浮点常量

浮点类型指定符：
十进制浮点常量
十六进制浮点常量 (C99)

浮点类型指定符：
float
double
long double (C89)
复数类型指定符 (C99)

浮点后缀：下列之一
f F l L

for表达式：
(初始分号$_{opt}$; 表达式$_{opt}$; 表达式$_{opt}$)

for语句：

for for表达式 语句

函数调用：

后缀表达式(表达式列表$_{opt}$)

函数声明器：

直接声明器(参数类型列表) (C89)

直接声明器(标识符列表$_{opt}$)

函数定义：

函数定义指定符 复合语句

函数定义指定符：

声明指定符$_{opt}$ 声明器 声明列表$_{opt}$

函数指定符：

inline (C99)

goto语句：

goto 名称标签;

h字符序列.

除了>和行末符之外的任何字符序列

十六进制常量：

0x 十六进制数字

0X 十六进制数字

十六进制数字常量 十六进制数字

十六进制浮点常量： (C99)

十六进制前缀 带点号的十六进制数字 二进制指数 浮点后缀$_{opt}$

十六进制前缀 十六进制数字序列 二进制指数 浮点后缀$_{opt}$

十六进制数字：下列之一

0 1 2 3 4 5 6 7 8 9

A B C D E F a b c d e f

十六进制数字序列：

十六进制数字

十六进制数字序列 十六进制数字

十六进制转义码：

x 十六进制数字

十六进制转义码 十六进制数字 (C89)

十六进制前缀：

0x

0X

十六进制四字组：

十六进制数字 十六进制数字 十六进制数字 十六进制数字

标识符：

非数字标识符

标识符 非数字标识符

标识符 数字

标识符列表：

标识符

参数列表，标识符

非数字标识符：

非数字

统一字符名称
其他由编译器所定义的字符

if - else语句：
if (表达式) 语句 else 语句

if语句
if (表达式) 语句

间接成员选择：
后缀表达式 -> 标识符

间接访问表达式：
* 类型转换表达式

初始分句：
表达式
声明 (C99)

初始化声明器：
声明器
声明器 = 初始化值

初始化声明器列表：
实始化声明器
实始化声明器列表，初始化声明器

初始化值：
赋值表达式
{ 初始化值列表 ，$_{opt}$ }

初始化值列表：
初始化值
初始化列表，初始化值
指定 初始化值 (C99)
初始化值列表，指定 初始化值 (C99)

整数常量：
十进制常量 整数后缀$_{opt}$
八进制常量 整数后缀$_{opt}$
十六进制常量 整数后缀$_{opt}$

整数后缀：
long后缀 unsigned后缀$_{opt}$
long long后缀 unsigned后缀$_{opt}$ (C99)
unsigned后缀 long后缀$_{opt}$
unsigned后缀 long long后缀$_{opt}$ (C99)

整数类型指定符：
有符号类型指定符
无符号类型指定符
字符类型指定符
布尔类型指定符 (C99)

迭代语句：
while语句
do语句
for语句

标签：
名称标签

```
            case标签
            default标签
带标签的语句：
            标签 : 语句
逻辑与表达式：
            位或表达式
            逻辑与表达式 && 位或表达式
逻辑反表达式：
            ! 类型转换表达式
逻辑或表达式：
            逻辑与表达式
            逻辑或表达式 || 逻辑与表达式
long long后缀：下列之一                                          (C99)
            ll  LL
long后缀：下列之一
            l  L
乘除表达式：
            类型转换表达式
            乘除表达式  乘除操作符  类型转换表达式
乘除操作符：下列之一
            *  /  %
名称标签：
            标识符
非数字：下列之一
            A  B  C  D  E  F  G  H  I  J  K  L  M
            N  O  P  Q  R  S  T  U  V  W  X  Y  Z
            a  b  c  d  e  f  g  h  i  j  k  l  m
            n  o  p  q  r  s  t  u  v  w  x  y  z
            _
非零数字：下列之一
            1  2  3  4  5  6  7  8  9
空语句：
            ;
八进制常量：
            0
            八进制常量 八进制数字
八进制数字：下列之一
            0  1  2  3  4  5  6  7
八进制转义码：
            八进制数字
            八进制数字  八进制数字
            八进制数字  八进制数字  八进制数字
打开-关闭开关：
            ON
            OFF
            DEFAULT
形参声明：
```

声明指定符 声明器
声明指定符 抽象声明器$_{opt}$

形参列表：
形参声明
形参列表，形参声明

形参类型列表：
形参列表
列参列表，…

带括号的表达式：
(表达式)

指针：
* 类型限定符列表$_{opt}$
* 类型限定符列表$_{opt}$ 指针

指针声明器：
指针 直接声明器

后缀减值表达式：
后缀表达式 --

后缀表达式：
基本表达式
下标表达式
成员选择表达式
函数调用
后缀增值表达式
后缀减值表达式
复合字面值 (C99)

后缀增值表达式：
后缀表达式 ++

前缀减值表达式：
-- 单目表达式

前缀增值表达式：
++ 单目表达式

预处理器标记：
任何C标记序列，或无法解释为标记的非空白字符序列，不是以<或"开头

基本表达式：
标识符
常量
带括号的表达式

q字符序列：
除了"和行末符之外的任何字符序列

关系表达式：
移位表达式
关系表达式 关系操作符 移位表达式

关系操作符：下列之一
< <= > >=

return语句：
return 表达式$_{opt}$;

s字符：

除了双引号"、反斜杠\或换行符之外的任何源字符
转义字符
统一字符名称 (C99)

s字符序列：
s字符
s字符序列 s字符

移位表达式：
加减表达式
移位表达式 移位操作符 加减表达式

移位表达式：下列之一
<< >>

有符号类型指定符：
short或short int或signed short或signed short int
int或signed int或signed
long或long int或signed long或signed long int
long long或long long int或signed long long或signed long long int

符号部分：下列之一
+ -

简单成员：
声明器

简单声明器：
标识符

sizeof表达式：
sizeof (类型名)
sizeof 单目表达式

语句：
表达式语句
标签语句
复合语句
条件语句
迭代语句
switch语句
break语句
continue语句
return语句
goto语句
空语句

存储类别指定符：下列之一
auto extern register static typedef

字符串常量：
" s字符序列$_{opt}$ "
L" s字符序列$_{opt}$ " (C89)

结构标签：
标识符

结构类型定义：
struct 结构标签$_{opt}$ { 字段列表 }

结构类型引用：

```
            struct 结构标签
结构类型指定符：
            结构类型定义
            结构类型引用
下标表达式：
            后缀表达式 [表达式]
switch语句：
            switch (expression ) 语句
顶层声明：
            声明
            函数定义
翻译单元：
            顶层声明
            翻译单元  顶层声明
typedef名称
            标识符
类型名称：
            声明指定符  抽象声明器opt
类型限定符：
            const
            volatile
            restrict                                          (C99)
类型限定符列表：                                              (C89)
            类型限定符
            类型限定符列表 类型限定符
类型指定符：
            枚举类型指定符
            浮点类型指定符
            整数类型指定符
            结构类型指定符
            typedef名称
            联合类型指定符
            void类型指定符
单目表达式：
            后缀表达式
            sizeof表达式
            单目负号表达式
            单目正号表达式
            逻辑反表达式
            位反表达式
            地址表达式
            间接访问表达式
            前缀增值表达式
            前缀减值表达式
单目负号表达式：
            - 类型转换表达式
单目正号表达式：                                              (C89)
```

+ 类型转换表达式

联合标签：

标识符

联合类型定义：

union 联合标签$_{opt}$ { 字段列表 }

联合类型引用：

union 联合标签

联合类型指定符：

联合类型定义

联合类型引用

统一字符名称：

\u 十六进制四字组

\U 十六进制四字组 十六进制四字组

无符号后缀：下列之一

u U

无符号类型指定符：

unsigned short int$_{opt}$

unsigned int$_{opt}$

unsigned long int$_{opt}$

unsigned long long int$_{opt}$ (C99)

void类型指定符：

void

while语句：

while (表达式) 语句

宽度：

常量表达式



附录C
练习答案

本附录包含了第2章～第9章的练习答案。

第2章的答案

1. 关键字、十六进制常量、宽字符串常量和括号是词法标记。注释和空格只能用于分隔标记。三字符组会在确认标记之前被去除。
2. 每个字符串的标记数量分别是：
 (a) 3个标记
 (b) 2个标记，-是操作符
 (c) 1个标记
 (d) 3个标记，第2个是“FOO”
 (e) 1个标记
 (f) 4个标记，**并不是一个单独的操作符
 (g) 没有标记，和“X\”相同，后者是个未终止的字符串常量
 (h) 没有标记，标识符不能使用$符号
 (i) 3个标记，*=是个操作符
 (j) 0个或3个。##并不是词法标记，但它恰好是个预处理器标记
3. 结果是***/；注释是由在下面的括号之间标识的。注释中的引号并不需要平衡。

```
/**/*/*"*/*/*"//*//**/*/
   (--) (---) (-----) (--)
```

4. 顺序是：
 (a) 转换三字符组
 (b) 处理行的延续
 (c) 删除注释
 (d) 把字符收集为标记
5. 原因和可能的解决方案：
 (a) 无法标识（阅读）标识符中的多个单词。使用大写字母或下或线。
 (b) 标识符的拼写与关键字太相似。
 (c) 小字字母l和大写字母O很容易和数字1和0混淆。
 (d) 很像一个数值常量（第1个字母是O）。
 (e) 如果编译器接受这个标识符，它就是使用了扩展。
6. (a) 例如：x = a // *divide*/ b。
 (b) 假设一个标准C编译器只区分标识符的前31个字符，第31个字符之后拼写不同的标识符被标准C认为是相同的，但在C++中却被认定为错误。

(c) 例如这个声明：int class = 0;

(d) 如果sizeof(char) != sizeof(int)，表达式sizeof('a') == sizeof(char)在C和C++中将是不同的。

第3章的答案

1. (a) 标准C或传统C并不允许在左括号之前出现空格。ident并不是带有1个参数的宏，而是一个没有参数的宏，被展开为“(x) x”。

 (b) =和;字符并无必要，并且很可能是错误的。在有些传统C编译器中，#后面的空格可能会产生问题。

 (c) 这个定义是正确的。

 (d) 这个定义是正确的，可以把关键字定义为宏。

2.

标准C	传统C
(a) b + a	b + a
(b) x 4（2个标记）	x4（1个标记）
(c) "a book"	# a book
(d) p?free(p):NULL	P?P?P?...:NULL:NULL:NULL（无限）

3. 预处理的结果是下面这3行（忽略空白）：

```
int blue = 0;
int blue = 0;
int red = 0;
```

4. 由于宏的参数和宏体没有加上括号，因此在大型表达式中宏的展开可能会被错误解释。更安全的定义是：

```
#define DBL(a)    ((a)+(a))
```

5. 这个宏是按照下面的步骤展开的：

```
M(M)(A,B)
MM(A,B)
A = "B"
```

6. 解决方案取决于defined和#error的存在：

```
#if !defined(SIZE) || (SIZE<1) || (SIZE>10)
#error "SIZE not properly defined "
#endif
```

7. 在预处理器命令#include </a/file.h>中，/a/file.h这个序列被看成是个标记（单个文件名），它对于编译器而言不是一个标记。

8. 假设程序员希望在x == 0时打印出一条错误信息。但是，x == 0是个运行时测试，则#error是条编译时命令。如果这个程序被编译，这条错误信息总是会出现，不管x的值如何，都会终止编译。

第4章的答案

1. 这个函数每次被调用时将返回它的实参的值。只有当P内部i的声明中 中用了存储类别static时，

后续调用的返回值才会发生变化。

2. f作为函数、整型变量、类型名称和枚举常量的声明都会发生冲突，最多只能存在1个。f既作为结构标签又作为联合标签会出现冲突；去掉这个联合，使f还被声明为一个结构成员。把f作为语句标签并不会与其他声明冲突（除了一些旧式的编译器）。

3.

```
    代码                        int i;        long i;       float i;
 1  int i;                      (声明)
 2  void f(i)                                 (声明)
 3     long i;                                (声明)
 4  {
 5     long l = i;                          (可使用的)
 6     {
 7        float i;                                          (声明)
 8        i = 3.4;                                         (可使用的)
 9     }
10     l = i+2;                             (可使用的)
11  }
12  int *p = &i;          (可使用的)
```

4.
```
(a) extern void P(void);
(b) register int i;
(c) typedef char *LT;
(d) extern void Q(int i, const char *cp);
(e) extern int R( double *(*p)(long i) );
(f) static char STR[11];(注意：保留null字符的空间)
(g) const char STR2[] = INIT_STR2;两边的花括号是可选的
    其他可接受的声明      const char *STR2=INIT_STR2;(没有花括号)
(h) int *IP = &i;
```

5.
```
int m[3][3] = {{1,2,3},{1,2,3},{1,2,3}};
```

第5章的答案

1. 注意这些类型都不应该涉及int，因为int类型的长度可能不比short更长。
 (a) long或unsigned long（unsigned short可能无法处理99999）
 (b) 包含2个成员的结构，两个成员分别是short类型（区号）和long类型（本地号码），也可以是这两种类型的无符号版本。
 (c) char（可以使用任何变型）
 (d) 标准C中的signed char，其他编译器中的short（char可能是无符号的）
 (e) 标准C中的signed char，其他编译器中的short（char可能是无符号的）
 (f) 可以使用double，但使用long类型并把余额按美分数量存储可以节省空间。
2. UP_ARROW_KEY的类型是int，它的值是0x86（134）。如果计算机使用signed类型表示char，扩展字符的值将是负的。因此，如果is_up_arrow的实参确实是0x86，return语句所执行的测试将是−122 == 134，结果是假而不是真。编写这个函数的正确方法是强迫字符码的类型为char，或者强迫实参的类型是unsigned char。也就是说，使用下面的return语句之一：

```
return c == (char) UP_ARROW_KEY;
return (unsigned char) c == UP_ARROW_KEY;
```

第一种解决方案可能更好，因为它允许更自由地定义UP_ARROW_KEY的值。

3. (a) 合法

 (b) 合法

 (c) 非法，无法解引用void *类型的指针

 (d) 非法，无法解引用void *类型的指针

4. (a) *(iv + i)

 (b) *(*(im_i)+j)

5. 13。类型转换在标准C中是必要的。在旧式的编译器中，它也可以使意思更为清晰，并且也可能是必要的。

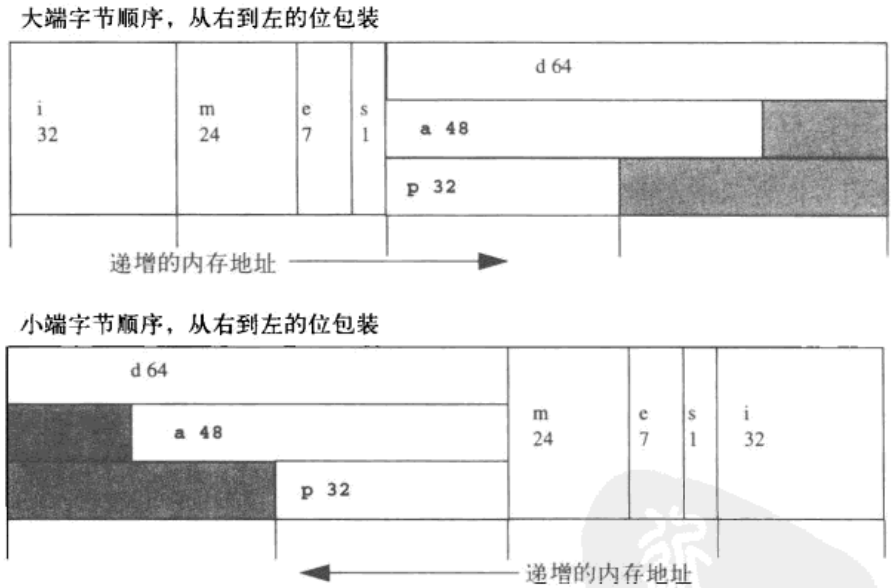
6. x.i = 0;

 x.F.s = 0;　　(0和1是仅有的合法值)

 x.F.e = 0; x.F.m = 0;

 x.U.d = 0.0; (或者x.U.p = NULL;，但不能是x.U.a[0] = '\0';，这将导致a的有些元素未定义)

7. 下面是草图。图中说明了每个字段所占据的位数，并在底部标注了字的边界。特别需要注意的是位段的宽度。



8.
```
typedef int *fpi(); /* type definition */
fpi *x; /* variable declaration */
int *fpi() /* function; can't use typedef in header */
{
    return (fpi *)0;
}
```

第6章的答案

1. 除了标准C不允许(c)和(e)之外，其他的类型转换在标准C和传统C中都是允许的。
2. 在解决方案中，我们假设传统C编译器允许混合指针赋值，在其他情况下则遵循标准C规则。但是，对于有些传统C编译器，答案将和问题1相同。
 (a) 标准C和传统C均允许
 (b) 标准C禁止，传统C允许
 (c) 标准C禁止，传统C允许
 (d) 标准C和传统C均允许
 (e) 标准C禁止，传统C允许
 (f) 标准C和传统C均允许
3. (a) unsigned
 (b) 传统C中的unsigned long或标准C中的long或unsigned long
 (c) double
 (d) 标准C的long double
 (e) int *（寻常单目转换作用于int []）
 (f) short (*)()，因为首先作用的是寻常单目转换。下面是一个可行的解决方案，将会发生这种情况：

```
extern short f1(), f2(), (*pf)();
extern int i;
pf = (i>0 ? f1 : f2);  /* binary conv on f1 and f2 */
```

4. 编译器可以使用32位表示char类型（虽然浪费）。不管表示形式如何，sizeof(char)的值始终是1。int类型的范围不能比char类型更小。它可以相同或者更大。
5. 它们之间并不一定要有任何关系。它们可以相同，也可以其中一个大于另一个。
6. 128这个值可以表示为32位的十六进制数00000080_{16}。由于计算机A采用了大端字节顺序，这些字节是按照00_{16}、00_{16}、00_{16}、80_{16}的顺序存储的。在小端字节顺序的计算机上，字节是从低端开始包装的，因此结果是80000000_{16}或−2 147 483 648。如果A采用小端字节顺序，而B采用大端字节顺序，结果是相同的。

第7章的答案

1. (a) char *
 (b) float（传统C是double）
 (c) float
 (d) int
 (e) float（传统C是double）
 (f) int
 (g) int

(h) int

(i) 非法

(j) float

2. (a) p1+=1; p2+=1; *p1=*p2;

(b) *p1=*p2; p1-=1; p2-=1;

3. (a) #define low_zeroes(n) (-1<<n)（如果n不大于int类型的宽度）

(b) #define low_ones(n) (~low_zeroes(n))

(c) #define mid_zeroes(width, offset) \

(low_zeroes(width+offset) | (low_ones(offset))

(d) #define mid_ones(width, offset (~mid_zeroes(width, offset))

4. 表达式j++ == ++j是合法的，但它的结果在标准C中是未定义的，因为j在同一个表达式中被修改了两次。取决于==的哪个操作数首先求值，其结果可以是0或1，尽管j的最终值可能是2。但是，j++&&++j是合法的，并具有正确的定义。它的结果是0。在表达式的最后，j的值是1。

5. (a) 允许，因为类型是兼容的

(b) 不允许（左边的引用类型的限定符不够充分）

(c) 允许，因为只有一种类型指定了长度

(d) 允许，因为右边不是左值，限定符无关紧要

(e) 不允许，仅仅是因为float与它提升后的类型double并不兼容

(f) 允许，因为引用类型是兼容的

6. 不。这个赋值是非法的，因为每个结构定义创建了一个新的类型。如果定义位于不同的源文件，这些类型是兼容的，但这只是为了允许多个片段有良好行为定义的程序顺利通过编译而采用的一种技术手段。

第8章的答案

1. (a)
```
n = A;
L1:
if (n>=B) goto L2;
sum+=n;
n++;
goto L1;
L2:;
```

(b)
```
L:
if (a<b) {
   a++;
   goto L;
}
```

(c)
```
L:
sum += *p;
if (++p < q) goto L;
```

2. j的值是3。“j是未定义的”这句话是不正确的。尽管程序的执行跳入到一个代码块，但i的存储类别是static，意味着它在程序开始执行之前就得到了合适的初始化。

3. sum的值是3。在每次执行循环时，i所接受的值分别是0、1、…。当i是0、1和3时，sum的值

增加1，continue语句使程序的执行进入循环的下一次迭代。当i是2时，sum的值并不会改变，但循环将会继续。但是，当sum为4时，switch中的break语句导致控制执行循环中的break语句，从而使循环终止。因此，case 5:永远不会被执行。

第9章的答案

1. (a) 合法的原型。
 (b) 合法的声明，但并不是原型；括号内必须有一个形参列表。
 (c) 非法的声明；在省略号之前至少要有1个形参声明。
 (d) 非法的声明；在每个形参名称前面至少有1个类型指定符、存储类别指定符或类型限定符。
 (e) 合法的原型；形参名称是不必要的。
 (f) 合法的定义，但并不是原型；括号内必须具有形参列表。
2. (a) 不兼容；原型的形参类型与寻常实参转换不兼容。如果定义并不是采用原型形式，这种兼容是必须的。
 (b) 不兼容；不管原型是否出现在定义中。
 (c) 兼容；形参名称并不需要相同。
 (d) 不兼容；两个原型在使用省略号方面并不一致。
 (e) 兼容；它们都是不是原型，因此所传递的是经过提升的实参类型。
 (f) 兼容。
3. (a) 非法，无法通过赋值转换把short *转换为int *。
 (b) 合法；s将转换为int类型，而ld保持不变。
 (c) 合法；ld将被转换为short类型。
 (d) 合法；第1个形参保持不变，第2个形参被转换为int类型，第3个形参保持不变。
 (e) 合法；形参在调用前被转换为int类型，并在被调用函数的开始处转换回short类型。
 (f) 合法，但很可能是错误的，形参不会变化，但将被调用者理解为int类型。
4. 这个调用受第1行所出现的原型的支配。后面这个声明并不会隐藏前者，因为P具有外部链接。
5. (a) 没问题。这个值在返回之前将被转换为short类型。
 (b) 没问题。这个值在返回之前将被转换为short类型。
 (c) 非法。这个表达式无法被转换为返回值的类型。
 (d) 非法。这个表达式无法通过赋值转换规则进行转换。

C语言经典译丛

C语言参考手册 原书第5版

C: A Reference Manual

(Fifth Edition)

这是一本C语言的权威参考手册，对C语言的基本概念和运行函数库提供了完整的描述，并强调了以正确性、可移性和可维护性为基本出发点的良好编程风格。

本书对C语言描述比其他任何书籍都要更加清晰和详细。本书涵盖的内容包括：

- 标准C（1999）：新版本的C标准支持复数型和布尔类型、可变长度数组、精确浮点编程以及为可移植性国际化所提供的新函数库。
- 标准C（1989）：当今大部分C程序员所使用的C语言版本。
- 传统C：1990年以前人们所使用的C语言。目前仍有数以百万行计的传统C代码还在使用中。
- C++和C的兼容：可以同时用于C和C++的代码。
- 适用于所有C语言版本的C运行时函数库。
- 本书对C的所有细节都进行了描述，是C语言编程人员和编译器实现者必备的参考手册。

本书经过深入的修订和更新，包含了对最新C语言标准（ISO/IEC 9899:1999）的完整描述，介绍了它强大的语言扩展和新函数库。

注意：网站http://www.CAReferenceManual.com提供了本书较长的示例程序的源代码、对C语言相关问题的深入讨论、最新的ISO/IEC语言标准修订以及其他重要C语言资源的链接。

上架指导：计算机/程序设计
ISBN 978-7-111-34684-5
9 787111 346845
定价：69.00元

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com
网上购书：www.china-pub.com

PEARSON
www.pearsonhighered.com

封面设计 范华明